Bachelor-Arbeit zur Erlangung des akademischen Grades
"Bachelor of Science (B.Sc.)"
im Studiengang Medizinische Informatik
an der
Medizinischen Fakultät der Universität Heidelberg
und der
Fakultät für Informatik der Hochschule Heilbronn

ENTWICKLUNG UND IMPLEMENTIERUNG EINES MARCHING CUBE BASIERTEN ALGORITHMUS ZUR PUNKTERHALTENDEN TRIANGULATION VON KONTUREN MIT SUBPIXEL-AUFLÖSUNG

FÜR VIRTUOS: EIN THERAPIEPLANUNGSSYSTEM FÜR DIE STRAHLENTHERAPIE

Referent: Prof. Dr. Rolf Bendl

Korreferent: Dr. Kristina Giske

Betreuer: Jan Rüppell

Thomas Sebastian Wollmann

Matrikel-Nr: 176111

Semester: 6

Charlottenstraße 26

74074 Heilbronn

Datum: 06.08.2013

Danksagung

Hiermit möchte ich mich bei allen herzlich bedanken, die mir die Durchführung dieser Arbeit ermöglicht haben.

Ich möchte mich bei meinem Referenten Herrn Prof. Dr. Bendl dafür bedanken, dass ich meine Arbeit am DKFZ in seiner Abteilung durchführen durfte.

Bei Dr. Kristina Giske bedanke ich mich für die Übernahme der Korreferenz.

Mein besonderer Dank gilt Jan Rüppell, der mich durch nette und kompetente Betreuung während der Arbeit begleitet hat.

Meinen Zimmerkollegen Markus Stoll und Michael Schwarz möchte ich für die zahlreichen konstruktiven Vorschläge und das tolle Arbeitsklima danken.

Der kompletten Arbeitsgruppe E0403 danke ich für ihre Hilfsbereitschaft.

INHALTSVERZEICHNIS

1.EINFÜHRUNG

1.1 AUFGABENSTELLUNG

Im Rahmen dieser Bachelorarbeit wurde ein Triangulations-Algorithmus basierend auf dem Marching Cube Algorithmus konzeptioniert und implementiert. Der Algorithmus soll in der Software für die Strahlentherapieplanung VIRTUOS verwendet werden und die speziellen Anforderungen dieser Software erfüllen. Diese sind im Kapitel 1.3 beschrieben.

Der Algorithmus soll eine Triangulation von Konturen mit Subpixel-Auflösung mit dem Marching Cube Verfahren ermöglichen. Diese Triangulation soll kanten- bzw. punkterhaltend sein. Somit darf sich die Position existierender Konturpunkte nicht ändern. Die Anzahl der erzeugten Dreiecke soll möglichst gering sein. Des Weiteren soll der Algorithmus in seiner Performanz optimiert sein.

Die Implementierung des in dieser Arbeit beschriebenen Algorithmus ist wie VIRTUOS in C/C++ geschrieben und soll die bisherige Delaunay Triangulation ablösen.

1.2 MOTIVATION

Die Triangulation in VIRTUOS dient hauptsächlich zur 3D-Visualisierung von Volumen. Zusätzlich wird sie zur Interpolation von Konturen und dem damit verbundenem Neuschneiden der Oberfläche zur Erzeugung nicht transversaler Konturen genutzt. Dies dient auch als Segmentierungshilfe für die manuelle Segmentierung von Strukturen.

Der bisherige Triangulations-Algorithmus [Har04] von VIRTUOS basiert auf der 3D-Delaunay Triangulation nach Boissonnat und Geiger [Boi93].

Die Delaunay Triangulation [Del34] versucht Dreiecke zu finden, die das Delaunay-Kriterium erfüllen [Har04]. Das Delaunay Kriterium besagt, dass innerhalb des Umkreises eines Dreiecks kein Punkt der Triangulation liegen darf [Har04]. Zum Finden solcher Dreiecke werden Voronoi-Diagramme verwendet [Har04].

Der aktuelle Algorithmus benötigt jedoch sehr lange für die Triangulation von Konturen, die sich überschneiden oder viele Punkte haben und stark irregulär geformt sind. Es ist sehr schwer abzuschätzen, wie viel Zeit die Delaunay Triangulation benötigt. Wenn sich die Form einer Kontur zwischen zwei Schichten stark ändert, kann keine Triangulation gefunden werden. Um solche Fälle zu lösen, mussten viele Sonderbehandlungen implementiert werden. Damit die Triangulation für die klinische Routine nutzbar ist, wurden für schwer triangulierbare Konturen Abbruchbedingungen implementiert.

Die Behandlung der Sonderfälle macht die Implementierung sehr unübersichtlich und mathematisch kompliziert. Des Weiteren können manche Triangulationen durch die Abbruchbedingungen gar nicht durchgeführt werden.

Die Verwendung des Marching Cube Ansatzes ist der Versuch, die oben genannten Probleme anzugehen. Der neue Triangulations-Algorithmus soll eine robustere und besser beherrschbare Triangulation ermöglichen.

1.3 ANFORDERUNGEN

Um die Erwartungen an den zu entwickelnden Algorithmus zu konkretisieren, wurden funktionale und nichtfunktionale Anforderungen definiert.

1.3.1 FUNKTIONALE ANFORDERUNGEN

1. Das Modul soll Konturen triangulieren. Somit aus Konturen Dreiecke generieren, die die durch die Konturen aufgespannte Außenfläche beschreiben.
2. Es sollen topologisch korrekte Dreiecksnetzmodelle erzeugt werden.
3. Das Modul soll Normalenvektoren auf den Dreiecken ausgeben, damit in späteren Schritten diese für Schattierung (Shading) verwendet werden können.
4. Das Modul soll die Anzahl der erzeugten Dreiecke möglichst sinnvoll reduzieren.
5. Die Punkte der gegebenen Konturen sollen nach der Triangulation in den Dreiecken als Vertices vorhanden sein.
6. Die Zuordnung von einem Punkt zu einer Schicht, zu einer Kontur und innerhalb der Kontur vor der Triangulation soll danach immer noch erhalten sein.
7. Dreiecke dürfen nur zwei benachbarte Schichten verbinden. Somit müssen auch alle Punkte des Dreiecksnetz auf Schichten liegen

1.3.2 NICHTFUNKTIONALE ANFORDERUNGEN

1. Das Modul soll komplett über ein Klasseninterface ansprechbar sein.
2. Der Algorithmus soll in seiner Performanz optimiert werden.
3. Die Einbindung in die VoiManSTL soll ähnlich zur Delaunay Triangulation sein.
4. Der Quellcode soll möglichst leicht erweiterbar und wartbar sein.
5. Der Algorithmus soll parametrisierbar sein um Performanz, Qualität der erzeugten Oberfläche und die Anzahl der erzeugte Dreiecke bzw. Vertices zu beeinflussen.

2.MATERIAL UND METHODEN

Dieser Abschnitt führt in die Literatur und die für das Verständnis dieser Arbeit wichtigen Themen ein. Es werden außerdem die zugrunde liegenden Algorithmen für den im Rahmen dieser Arbeit entwickelten Algorithmus näher erläutert.

2.1 ÄHNLICHE ARBEITEN

Für die Triangulation von medizinischen Bilddaten werden Algorithmen basierend auf dem Marching Cube Algorithmus (MC) [Lor87] neben den Algorithmen basierend auf der Delaunay Triangulation [Del34] häufig eingesetzt.

Der Marching Cube Algorithmus arbeitet nach dem „divide and conquer“ Paradigma und teilt den $\mathbb{R}^3$ in einzelne zu betrachtende Quader (Voxel) auf. Diese werden mittels einer Lookuptable (LUT) trianguliert und danach anhand der Voxel- bzw. Isowerte interpoliert. Die Lookuptable beinhaltet alle möglichen Kombinationen aus Ecken und resultierenden Dreiecken. Jedoch sind alle Konfigurationen auf 15 Grundkonfigurationen zurückzuführen, die in gedrehter oder gespiegelter Form mehrfach vorhanden sind. Eine ausführlichere Erklärung des Marching Cube ist in Kapitel 2.2.4 zu finden.

Es gibt jedoch mehrdeutige Konfigurations-Fälle, die zu Löchern in der triangulierten Oberfläche führen können. Es gab zahlreiche Bestrebungen diese uneindeutigen Konfigurationen zu lösen. Der bekannteste Ansatz ist der Marching Cube 33 (MC33) [Evg95]. Dieser erweitert die ursprünglichen Grundkonfigurationen um 18 weitere Konfigurationen auf 33. Der MC33 wird in der Literatur oft als Standard Marching Cube bezeichnet.

Weitere in der Literatur beschriebene Ansätze versuchen durch geeignete Datenstrukturen wie einem Octree die Performanz zu verbessern [Wil92]. Eine andere Möglichkeit die Geschwindigkeit zu erhöhen ist das Parallelisieren des Algorithmus auf der CPU [Tim04] oder der GPU [FRe04].

Der Marching Cube Algorithmus erzeugt grundsätzlich viele Dreiecke. Zur Beschreibung der Oberfläche ist aber nur ein Bruchteil von diesen Dreiecken nötig. Deswegen wird ein Dreiecks-Dezimierungsalgorithmus, wie das Verfahren nach Schroeder et al. [WJS92], nachgeschaltet. Dezimierungsverfahren können auch parallelisiert werden, wie der Ansatz von Shontz et al. [Sho13] für eine CPU-GPU Implementierung eines Dezimierungsalgorithmus zeigt. Der Diskrete Marching Cube (DiscMC) [Mon94] versucht schon während der Triangulierung geschickt Dreiecksflächen zusammenzufassen und damit Speicher und Zeit zu sparen. Somit ist eine nachträgliche Dezimierung nicht mehr unbedingt nötig.

Desweiteren gibt es viele Arbeiten zur Oberflächenverbesserung. Eine gängige Methode ist das Glätten des erzeugten Dreiecksnetzes mit z.B. der Dreiecksnetz-Glättung nach Laplace [Her76] oder dem Humphrey's Classes Algorithmus [Bru99]. Eine genauere Erklärung der Dreiecksnetz-Glättung nach Laplace befindet sich in Kapitel 2.2.6. Eine weitere Möglichkeit die Oberfläche und Performanz zu verbessern ist es ein adaptives Verfahren für die Aufteilung des Volumens in Voxel zu verwenden [Shu95]. Dadurch werden Flächen mit starker Krümmung detailreich trianguliert. Flächen mit wenig Krümmung können durch große Dreiecke dargestellt werden.

Damit aus Konturstapeln ein für den Marching Cube triangulierbarer Voxelraum wird, werden in der Literatur unterschiedliche Voxelisierer verwendet. Ein verbreitetes Verfahren ist das mittels Euklidischen Distanzfeldern [Bra06]. Dieses Verfahren bietet zusätzlich die Möglichkeit, die gegebenen Konturen zu interpolieren.

Eine relativ neue Marching Cube Abwandlung ist der Simplified Pattern Marching Cube (spMC) [Shi08]. Er zeichnet im Gegensatz zum Standard Marching Cube Kanten nicht an den Außenflächen, sondern mittig vom Voxel. Die Abweichung bzw. der Fehler der Approximation ist laut Shi et al. vernachlässigbar, da die verwendeten Datensätze hochauflösend sind. Sie erzeugt jedoch deutlich weniger Dreiecke und ist durch die kleinere Anzahl von Fällen und Dreiecken performanter und robuster gegen Gaußschesrauschen und Schrotrauschen als der Standard Marching Cube. Im Kapitel 3.1 werden weitere Vorteile vom Simplified Pattern Marching Cube für unseren Anwendungsfall besprochen.

2.2 GRUNDLAGEN

In diesem Abschnitt erfolgt die Vorstellung der Grundlagen, die zum Verständnis dieser Arbeit wichtig sind. Es werden einzelne Aspekte von VIRTUOS näher beleuchtet und die grundlegenden Algorithmen, zum Verständnis des im Rahmen dieser Arbeit entwickelten Algorithmus, erläutert. Erklärungen zu weiteren Begriffen sind in Kapitel 8.1 zu finden.

2.2.1 VIRTUOS ALLGEMEIN

VIRTUOS (**VIRTU**al Radi**O**therapy **S**imulator) [Ben93] ist eine Strahlentherapie-Simulations-Software des Deutschen Krebsforschungszentrum (DKFZ). Sie bietet das grafische Frontend für das aus dem gleichen Hause stammende Strahlentherapie-Planungssystem VOXELPLAN. Sie entstand im Rahmen einer Diplomarbeit von Prof. Dr. Rolf Bendl [Ben91] und wird seitdem im DKFZ in der Abteilung für Medizinische Physik in der Strahlentherapie weiterentwickelt.

VIRTUOS bietet die Möglichkeit einen dreidimensionalen Therapieplan zu erstellen, zu optimieren und zu evaluieren. Dafür werden weitere Tools, wie das auch aus dem DKFZ stammende KonRad (**Kon**formiertende **Rad**iotherapie) [Bor06], verwendet.

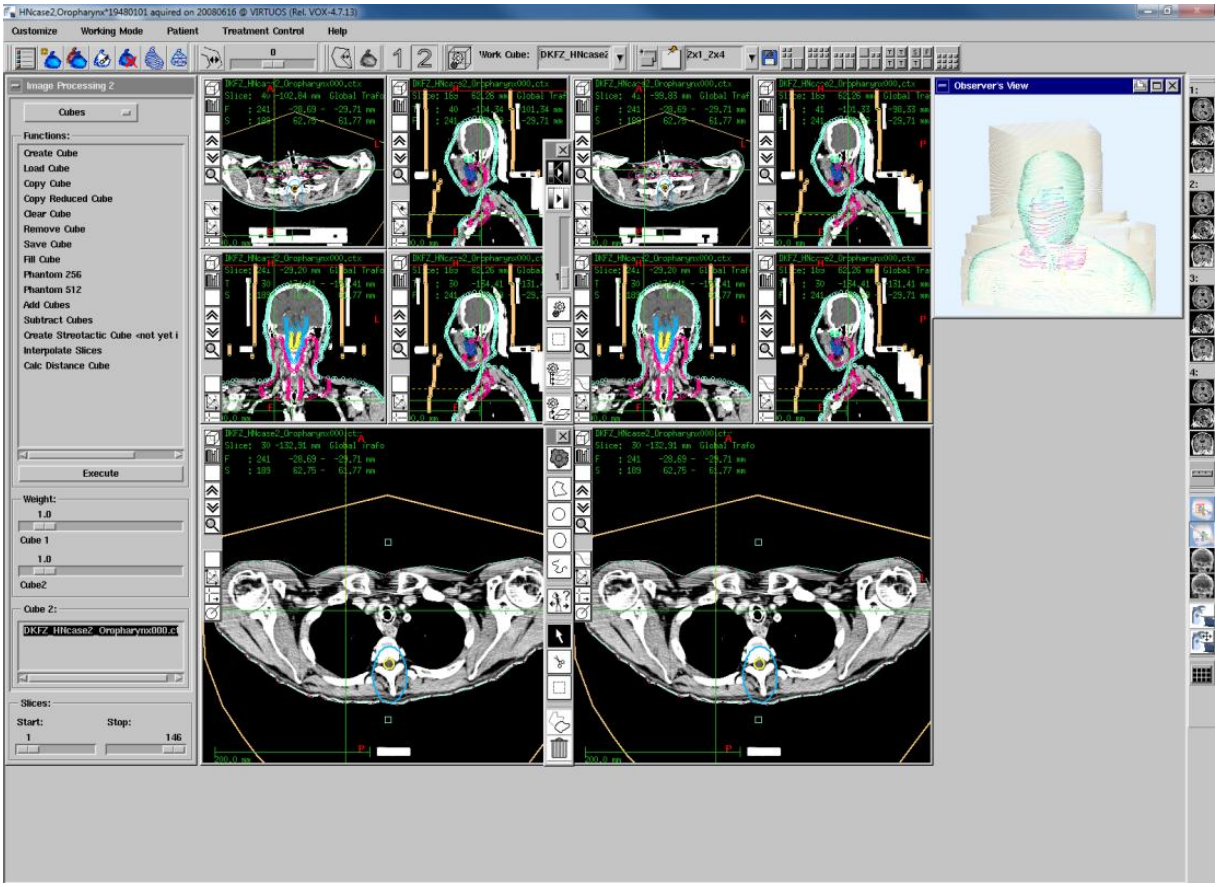

Abbildung 1: Bedienoberfläche von VIRTUOS

VIRTUOS kann in mehreren Arbeitsmodi betrieben werden. Es gibt den „Image Processing“, den „Planung“, den „Stereotaxie“ und den „Result“ Modus. Der für diese Arbeit wichtige Modus ist der „Image Processing“ Modus.

Im „Image Processing“ Modus kann der Nutzer Strukturen definieren. Diese sogenannten Volume of interest (Voi) werden genutzt um Risikostrukturen und Zielvolumina für die Bestrahlungsplanung zu definieren.

Das Festlegen einer Voi erfolgt über Konturen. Dafür definiert der Nutzer, manuell oder Algorithmen gestützt, in CT-Schichten Konturen. Abbildung 2 zeigt eine eingezeichnete Kontur im „Image Processing“ Modus von VIRTUOS.

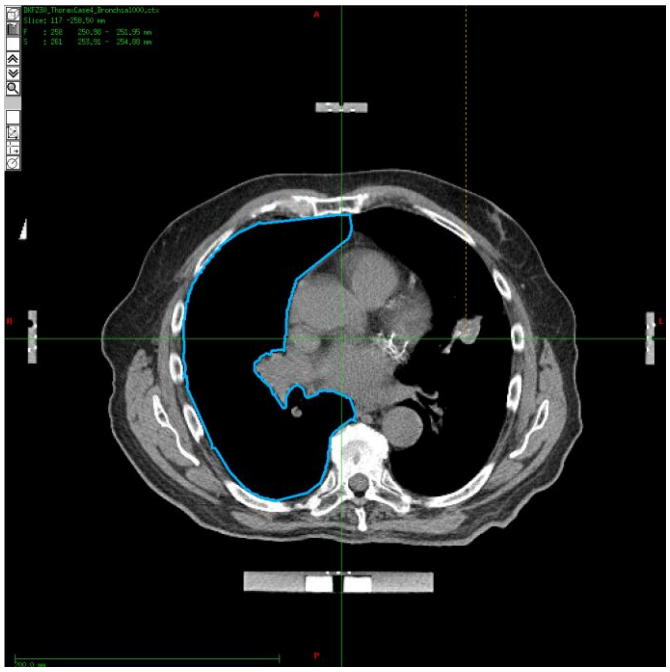

Abbildung 2: eingezeichnete Kontur in VIRTUOS

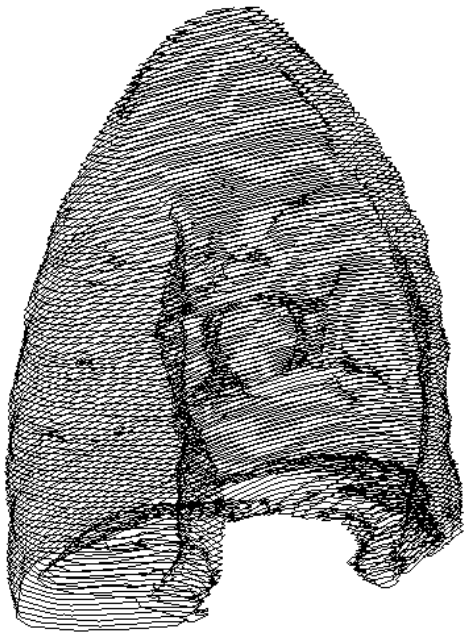

Abbildung 3: Lungenflügel dargestellt als Konturstapel

Mehrere Konturen in einer Voi ergeben einen Konturstapel, wie in Abbildung 3 zu sehen ist. Dieser definiert somit das von der Struktur eingenommene Volumen. Die Darstellung der Voi in 3D erfolgt in Virtuos zuerst nur mittels Konturen, in einem sogenannten Bändermodell. Zusätzlich kann der Nutzer, um sich einen besseren Eindruck von der Struktur zu verschaffen, das Volumen trianguliert als Oberfläche anzeigen lassen.

Der Quellcode von VIRTUOS bzw. VOXELPLAN ist in Module aufgeteilt. Ein für diese Arbeit wichtiges Modul ist die VoiManSTL, welches in Kapitel 2.2.3 vorgestellt wird. Der in dieser Arbeit entwickelte Algorithmus wird als eigenständiges Modul in VIRTUOS integriert.

2.2.2 VIRTUOS KOORDINATENSYSTEME

In VIRTUOS gibt es für verschiedene Anwendungsfälle unterschiedliche Koordinatensysteme. Die für diese Arbeit Wichtigen werden in diesem Kapitel vorgestellt.

Das CT-Koordinatensystem wird vom CT-Scanner festgelegt. Es ist in Millimeter und die X-Achse ist parallel von der rechten zur linken Seite des Patienten. Die Y-Achse zeigt orthogonal von der ventralen zur dorsalen Seite des Patienten. Die Z-Achse zeigt orthogonal zur X- und Y-Achse von kaudal nach kranial. Der Ursprung liegt in der unteren vorderen linken Ecke in kranialer Richtung des Patienten im CT-Datensatz. Voraussetzung dafür ist, dass der Patient Head First Supine (HFS) oder Feed First Supine (FFS) im CT liegt.

Das Array-Koordinatensystem definiert ein Koordinatensystem in einem 3D-Array. Die Ausrichtung der Achsen ist wie im CT-Koordinatensystem. Der Index im Array-Koordinatensystem gibt die Position eines spezifischen Voxels in der CT-Bildserie an. Die Maxima der X und Y Werte sind gegeben durch die Auflösung der CT-Schichtbilder. Das Maximum für den Z-Wert ist gegeben durch die Anzahl der vorhandenen CT-Schichten.

Das Würfel-Koordinatensystem (Cube-Koordinatensystem) ist analog zum Array-Koordinatensystem mit dem Unterschied, dass ein Index im Würfel-Koordinatensystem als float Wert angegeben ist. Dadurch kann ein Punkt in Subvoxel-Auflösung definiert werden. Die X- und Y-Koordinate eines Voxels ist außerdem abweichend auf der Mitte eines Voxels definiert. Die Z-Koordinate entspricht der Schichtposition vom CT-Scanner.

Abbildung 4 demonstriert wie ein Patient, in HFS Position, im Würfel-Koordinatensystem liegt.

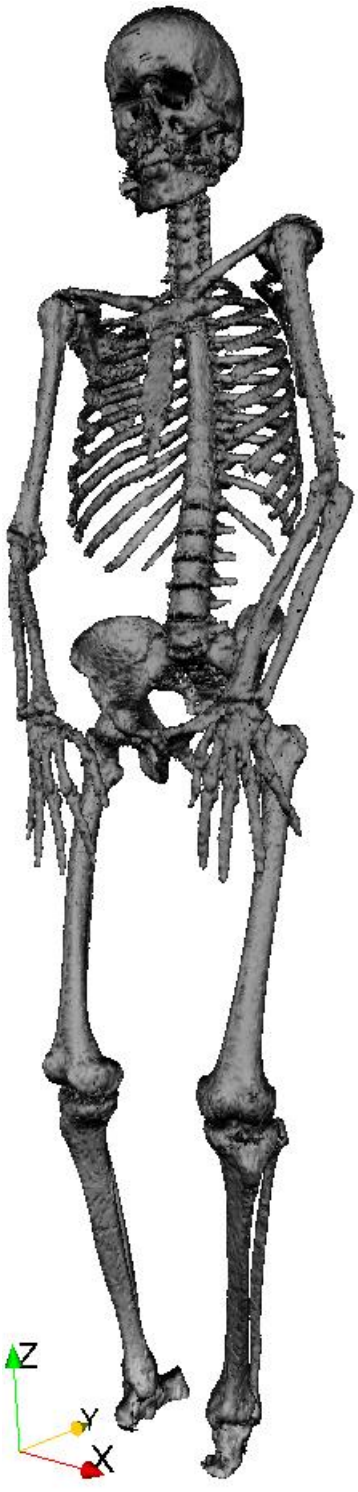

Abbildung 4: Skelett im Würfel-Koordinatensystem

2.2.3 VoiManSTL

Das wichtigste Modul von VIRTUOS für die Integration des im Rahmen dieser Arbeit entstandenen Algorithmus ist die Volume of interest Manager Standard Library (VoiManSTL). Diese verwaltet das Patientenmodell. Es entstand im Rahmen einer Diplomarbeit von Hans-Jörg Fischer [Han94]. Abbildung 5 zeigt die grundlegende Klassenstruktur des Moduls.

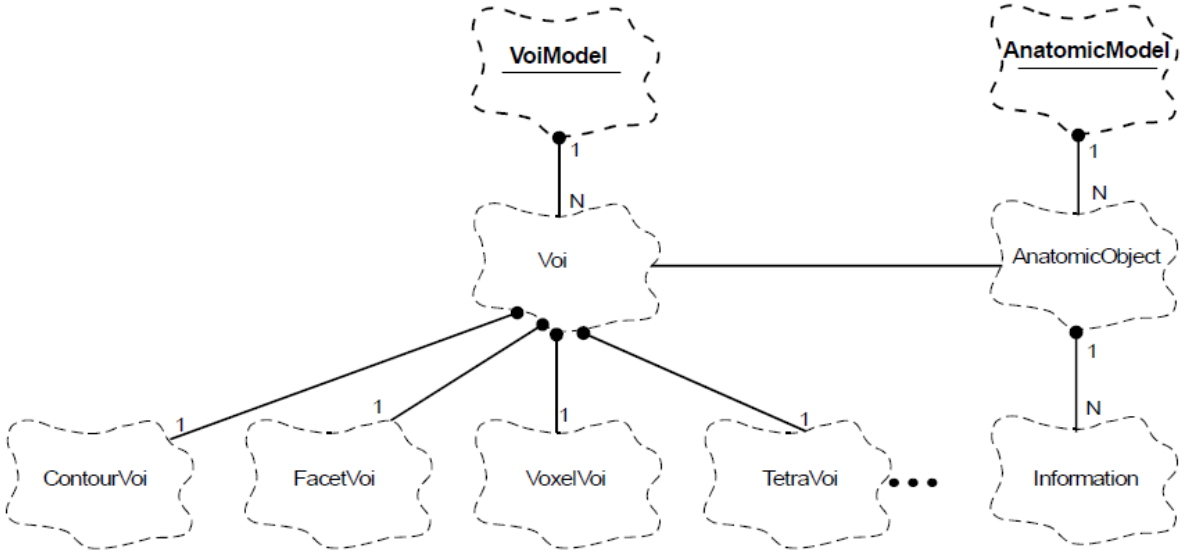

Abbildung 5: grundlegende Klassenstruktur der VoiManSTL [Han94]

Das Patientenmodell ist das *AnatomicModel*. In einem *AnatomicModel* gibt es mehrere Objekte die *AnatomicObjects*. Analog dazu gibt es ein *VoiModel*, das die für dieses Modell interessanten Strukturen in Form von *Voi* Objekten hält. Eine Voi-Struktur kann in verschiedenen Darstellungen vorliegen. Die gleichnamige Klasse hält alle geometrischen Repräsentationsformen jeweils einmal [Han94].

Die *VoxelVoi* besteht analog zum Namen aus Voxeln. Die *ContourVoi* verwaltet die Konturen und bildet den Konturstapel. Die *FacetVoi* beinhaltet die triangulierte Oberfläche der Voi. Für biomechanische Simulationen werden oft mit Tetraedern gefüllt Volumen genutzt. Die Klasse *TetraVoi* bietet die theoretische Möglichkeit die Voi mit Tetraedern darzustellen.

Die Struktur der Klasse *FacetVoi* ist besonders wichtig für das Verständnis, wie ein Dreiecksnetz in Virtuos repräsentiert wird. Sie besteht aus mehreren Schichten, den *FacetSlices*. Diese verbinden jeweils zwei benachbarte Konturschichten. Jede *FacetSlice* hält dafür Listen mit Dreiecken, die auf Punkte in der *ContourVoi* verweisen.

2.2.4 MARCHING CUBE

In Kapitel 2.1 wurde die Funktionsweise des Marching Cube Algorithmus schon ansatzweise erklärt. Da er jedoch die Grundlage für diese Arbeit darstellt, wird er in seiner Grundform [Lor87] in diesem Abschnitt noch einmal ausführlich vorgestellt.

Der Marching Cube Algorithmus ist ein Algorithmus um eine Isofläche S_c eines Objekts, das in einem Skalarfeld $\varphi: \mathbb{R}^n \to \mathbb{R}$ beschrieben wird durch Dreiecke zu approximieren. Der Isowert $c \in \mathbb{R}$ beschreibt die gemeinsame Eigenschaft des Objekts wie z.B. gleiche Dichte, Temperatur oder emittierter Strahlung.

Für eine Isofläche $S_c := \{v \in \mathbb{R}^n | \varphi(v) = c\}$ wird durch den Algorithmus eine endliche Menge von Datenpunkten $P \subset \mathbb{R}^n \times \mathbb{R}$ zur Approximation von S_c erzeugt [Han05].

Der Marching Cube wird üblicherweise im $\mathbb{R}^2$ und im $\mathbb{R}^3$ verwendet. Im $\mathbb{R}^2$ nennt man ihn den Marching Square und er dient in der Literatur häufigt zur Demonstration eines neuen Verfahrens. Im Folgenden wird der Marching Cube Algorithmus im $\mathbb{R}^3$ beschrieben.

Der Marching Cube Algorithmus verwendet einen logischen Würfel, der über ein gegebenes Voxelgitter geschoben wird. Die Eckpunkte des Würfels befinden sich in den Mittelpunkten der acht benachbarten Voxel zweier Schichten [Ben13]. Dies ergibt jeweils eine Konfiguration. Die zugehörigen Dreiecke werden aus einer vorberechneten Lookuptable (LUT) gelesen und danach anhand der Voxelintensitäten interpoliert. Der konkrete Ablauf ist in der nebenstehenden Abbildung 6 zu sehen.

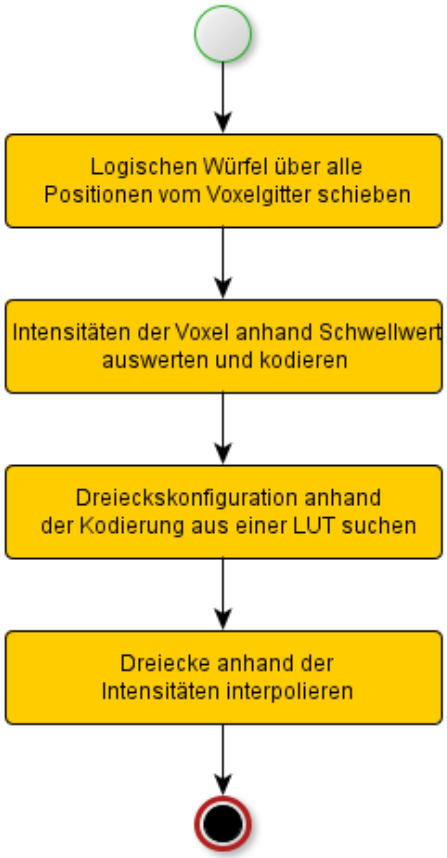

Abbildung 6: Ablauf des Marching Cube

Anhand des Schwellwertes kann jeder der acht Voxel zum Objekt gehören oder nicht. Dies führt dazu, dass es 2^8 verschiedene Voxelanordnungen geben kann. Dadurch, dass viele der 256 Möglichkeiten durch Rotationen und Spiegelungen ineinander überführt werden können, gibt es eigentlich nur 15 unterschiedliche Konfigurationen. Die Abbildung 7 zeigt alle 15 Grundkonfigurationen.

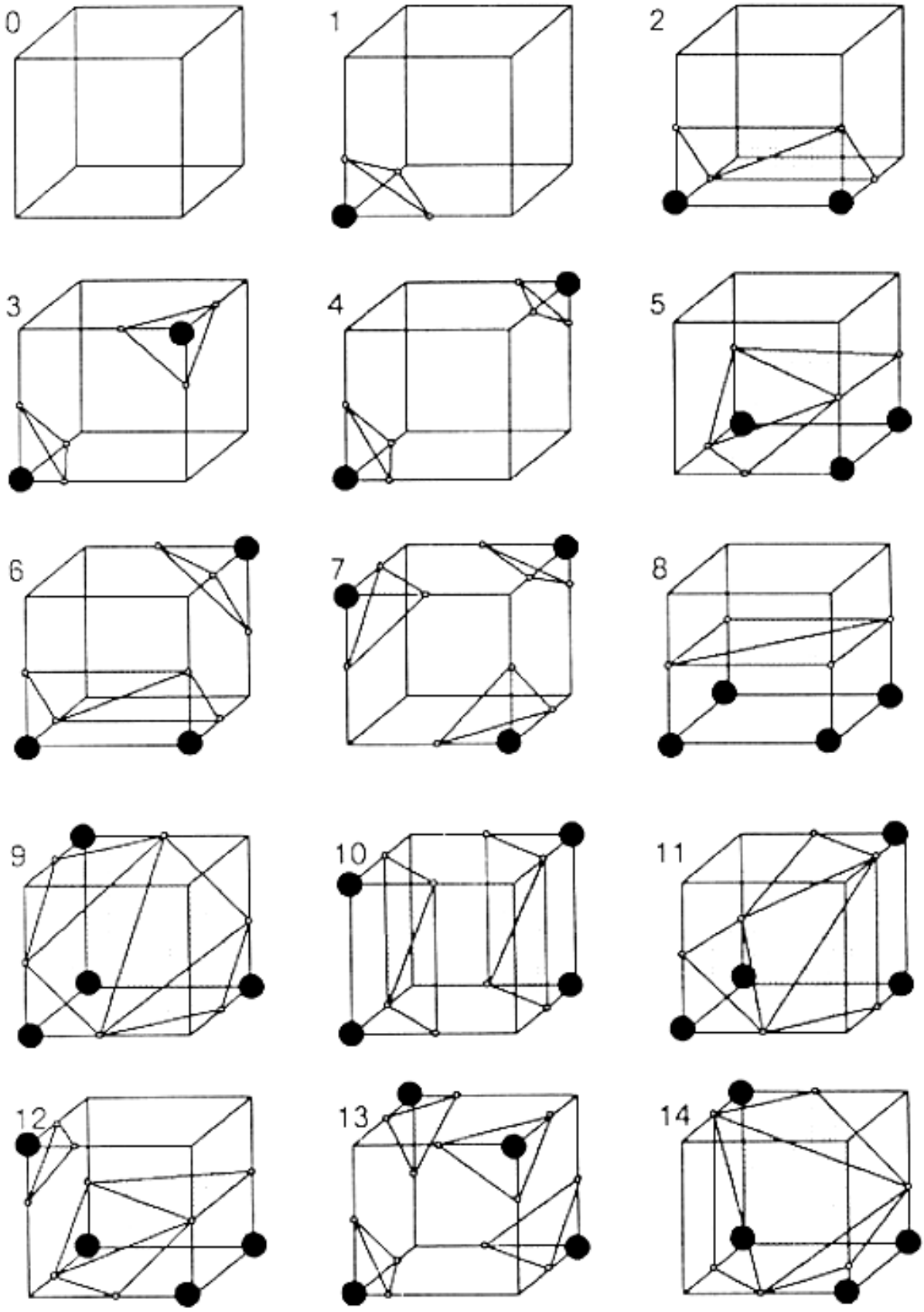

Abbildung 7: Marching Cube Grundkonfigurationen [Lor87]

2.2.5 Dreiecksnetz Dezimierung nach Schroeder et al.

Der Marching Cube Algorithmus erzeugt prinzipbedingt eine sehr hohe Anzahl von Dreiecken. Diese können aus Performanz Gründen durch ein Dezimierungsverfahren verringert werden, ohne die Qualität der Oberfläche maßgeblich zu beeinträchtigen. Ein bekanntes Verfahren ist das nach Schroeder et al. [WJS92]. Der im Rahmen dieser Arbeit entwickelte Dezimierungsalgorithmus basiert auf diesem Verfahren und wird deswegen in folgendem Kapitel vorgestellt.

Die Aufgabe eines Dezimierungs-Algorithmus ist die Anzahl der Dreiecke, die nötig sind eine Oberfläche zu beschreiben, möglichst sinnvoll zu reduzieren, doch gleichzeitig die Originaltopologie durch eine gute Approximation zu erhalten.

Der Ansatz von Schroeder et al. charakterisiert im ersten Schritt jeden Vertex anhand seiner Geometrie und Topologie. Daraufhin werden die zu reduzierenden Kandidaten mittels eines Dezimierungskriteriums ausgesucht. Die gefundenen Vertices werden entfernt und das entstandene Loch neu trianguliert.

Schroeder et al. teilen jeden Vertex in die unten aufgelisteten fünf Kategorien ein [WJS92].

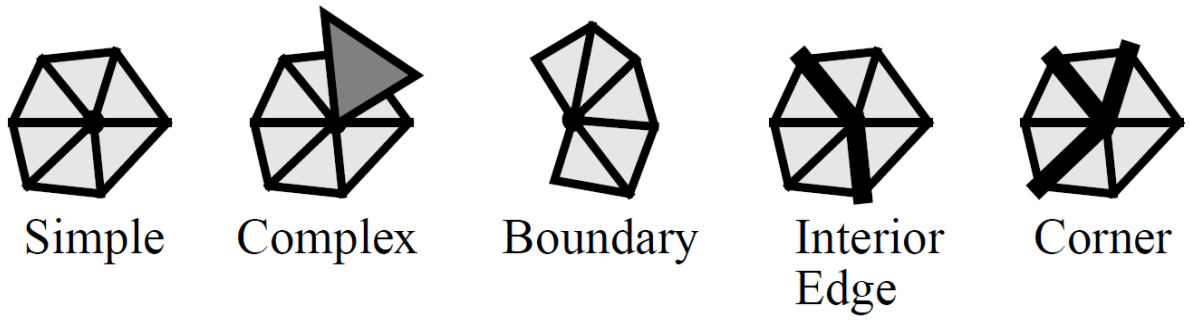

Abbildung 8: Vertex Kategorien nach Schroeder et al. [WJS92]

Simple Vertices beschreiben planare Flächen. Verwand mit ihnen sind die Corner Vertices, die Ecken im Dreiecksnetz beschreiben. Boundary und Interior Edges beschreiben Kanten am Rand bzw. innerhalb des Dreiecknetzes. Alle anderen Vertices gehören zu den Complex Vertices.

Wichtig für den in diesem Kapitel beschriebenen Algorithmus sind alle Ecktypen mit Ausnahme der Complex Vertices. Diese werden durch den Algorithmus von Schroeder et al. nicht dezimiert.

Simple und Corner Vertices werden mittels des „Kriteriums der mittleren Ebene“ überprüft. Dabei wird anhand des geometrischen Schwerpunkts der benachbarten Dreiecke eines Vertex eine mittlere Ebene errechnet. Diese Ebene ist in Abbildung 9 zu sehen. Wenn der Abstand **d** von Vertex zur Ebene kleiner als ein festgelegter Schwellwert ist wird der Vertex dezimiert. Die Fläche bei einem Simple Vertex ist immer planar. Somit ist in diesem Fall **d** immer gleich null.

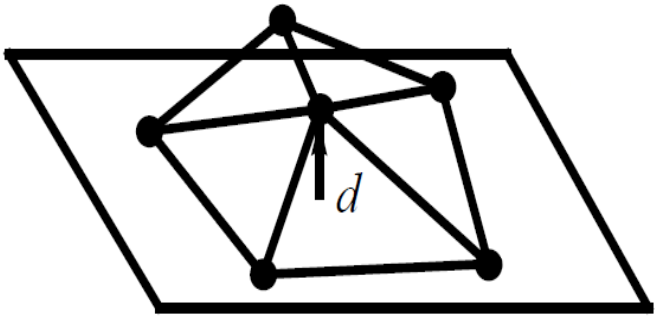

Abbildung 9: Kriterium der mittleren Ebene [WJS92]

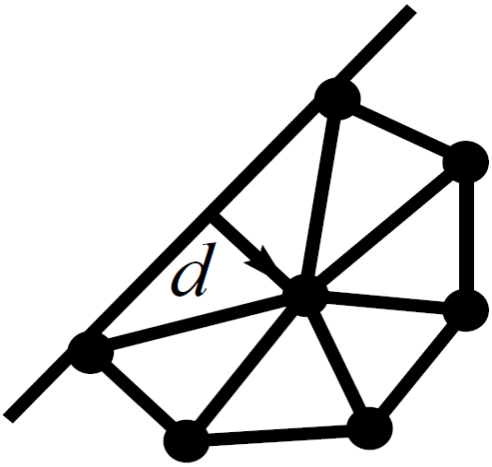

Abbildung 10: Abstand zu einer Linie Kriterium [WJS92]

Für Boundary und Interior Edges gibt es stattdessen das „Abstand zu einer Linie Kriterium“. Dabei wird jeweils zwischen allen Nachbarn des Vertex eine Gerade gespannt und der Abstand **d** vom Vertex zu dieser Linie als Kriterium für die Dezimierung genommen. Abbildung 10 zeigt das Kriterium anhand eines Boundary Edge Vertex.

Die daraus resultierenden Kandidaten können nun dezimiert und neu trianguliert werden.

Für die eigentliche Dezimierung bzw. Neutriangulierung verwendet der Algorithmus eine rekursive Schleifentriangulation mittels einer Spaltebene.

Da dieses Triangulierungs-Verfahren in dieser Arbeit nicht verwendet wird, wird dieses Verfahren hier nicht weiter erläutert. Das in dieser Arbeit zur Dreiecksnetzdezimierung verwendete Verfahren wird in Kapitel 3.5.3 beschrieben.

Es gibt Fälle in denen eine Triangulation fehl schlagen kann. Dies hängt vom verwendeten Verfahren ab. Wenn dies geschieht wird der Punkt nicht dezimiert.

In Abbildung 11 ist der Ablauf des Dreiecksnetz Dezimierungsalgorithmus nach Schroeder et al. nochmal visuell beschrieben.

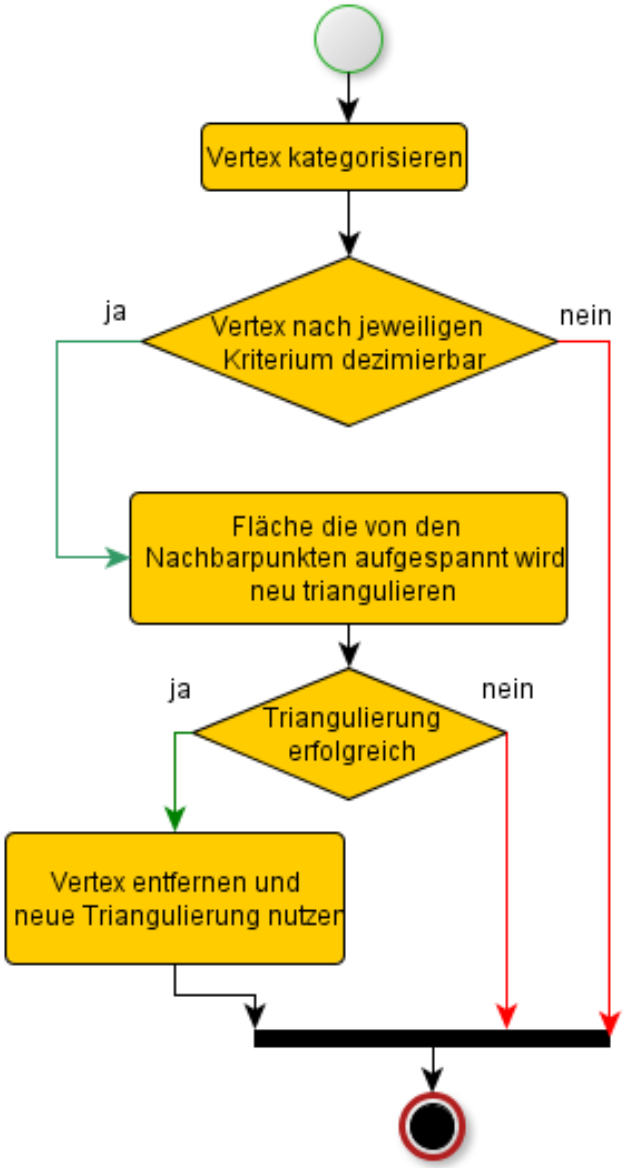

Abbildung 11: Ablauf des Dreiecksnetz Dezimierungsalgorithmus nach Schroeder et al.

2.2.6 Edge-Collapse

Ein in der Literatur häufig verwendetes Verfahren zum Dezimieren von Dreiecken in einem Dreiecksnetz ist der Edge Collapse [Gar97]. Der Edge Collapse beschreibt das Zusammenführen zweier Vertices zu einem und gehört somit zu den High Level Euler Operatoren [Hop93]. Bei dieser Dezimierung kollabieren immer zwei Dreiecke und ein Vertex wird dezimiert. Diese Methode wird nochmal unterteilt in den Half-Edge- und den Full-Edge Collapse. Beim Half-Edge Collapse wird ein Vertex logisch an die Position des Anderen verschoben und dort zu Einem vereint. Der Full-Edge Collapse verschiebt beide Vertices in die Mitte der Beiden und vereinigt sie dort.

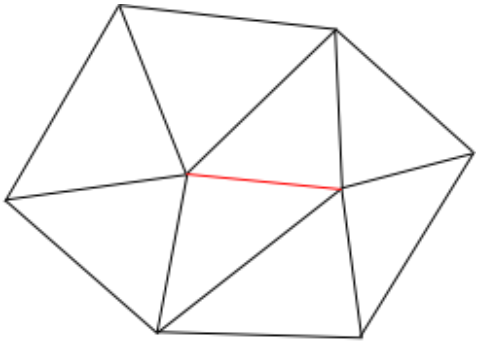

Abbildung 12: Vertices vor Edge Collapse

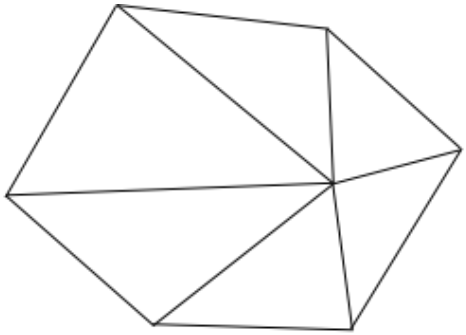

Abbildung 13: Half-Edge Collapse

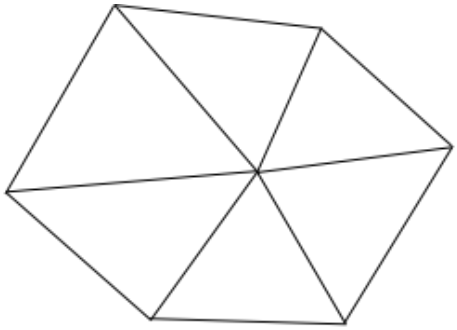

Abbildung 14: Full-Edge Collapse

2.2.7 Laplace Dreiecksnetz Glättung

Der Marching Cube Algorithmus erzeugt ohne Interpolierung sehr kantige Oberflächen. Die Interpolierung mittels der Isowerte ermöglicht eine gute Approximation der Oberfläche. Zur weiteren Optimierung wird oft ein Glättungsfilter verwendet. In der Literatur wird häufig der Humphrey's Classes Algorithmus [Bru99] verwendet. Er basiert auf der Laplace Dreiecksnetz Glättung [Her76], die hier kurz vorgestellt wird, da sie für diese Arbeit relevant ist.

Die Laplace Glättung entspricht dem Laplace Operator in dem Fall, dass das Dreiecksnetz ein rechteckiges Netz ist, in dem jeder Vertex vier Nachbarn hat [Her76]. Die Laplace Dreiecksnetz Glättung errechnet für jeden Vertex x_i, mit N Nachbarn, in einem Dreiecksnetz eine neue Position $\bar{x}_i$ anhand der Position der Nachbarn x_1 bis x_N. Sie ist wie folgt definiert.

$$\bar{x}_i = \frac{1}{N}\sum_{j=1}^{N} x_j$$

Formel 1: Laplace Dreiecksnetz Glättung

3.ERGEBNISSE

Dieser Abschnitt befasst sich mit den im Rahmen dieser Arbeit entstandenen Resultaten. Er beinhaltet den Entwurf des im Rahmen dieser Arbeit entstandenen Algorithmus. Außerdem werden nennenswerte Besonderheiten der Implementierung erläutert. Danach wird die Integration in VIRTUOS näher beleuchtet. Der Abschnitt schließt mit der Evaluation des entwickelten Algorithmus ab.

3.1 WAHL DER ALGORITHMEN

Zuerst erfolgte ein Literaturstudium und eine Bewertung der bekannten Marching Cube basierten Algorithmen. Nach dem Vergleich von verschiedenen Marching Cube Derivaten fiel die Entscheidung auf den Simplified Pattern Marching Cube (spMC) [Shi08]. Er wurde in Kapitel 1.2, in seiner ursprünglichen Bestimmung, vorgestellt. In diesem Kapitel wird erklärt, warum der Algorithmus für diesen Anwendungsfall besonders geeignet ist.

Der Standard Marching Cube ist für die Anwendung auf Volumen-Daten gedacht. Durch die verwendete Lookuptable (LUT) werden die generierten Flächen an Voxelkanten gezeichnet, an denen ein benachbartes Voxel einen Schwellwert überschreitet und das andere Benachbarte ihn unterschreitet. Diese Eigenschaft entspricht einer implizierten Segmentierung während der Triangulierung. In dieser Arbeit soll jedoch eine Oberflächenvisualisierung anhand von Konturen erfolgen. Damit das Marching Cube Prinzip angewandt werden kann, müssen die Konturen in einen Voxelraum eingezeichnet und gefüllt werden (voxelisiert). Wenn man annimmt, dass der Algorithmus jedes Voxel markiert, dass die gegebene Kontur kreuzt, dann zusätzlich die eingeschlossenen Voxel markiert und daraufhin den Marching Cube ausführt, dann liegt die triangulierte Fläche um bis zu ein Voxel neben der Ursprünglichen. Dieser Effekt ist unerwünscht, wenn eine Oberfläche benötigt wird, in der die Konturpunkte wieder vorkommen müssen. Das folgende Beispiel, bei dem eine Kontur (rot) erst voxelisiert bzw. in 2D diskretisiert (weiß) (Abbildung 15) und dann mit dem Marching Square wieder rekonstruiert wird (Abbildung 16), demonstriert das Problem.

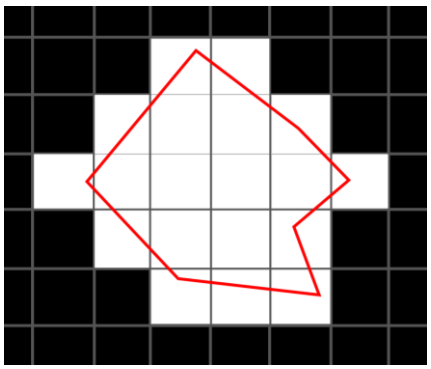

Abbildung 15: Beispiel Kontur Diskretisierung

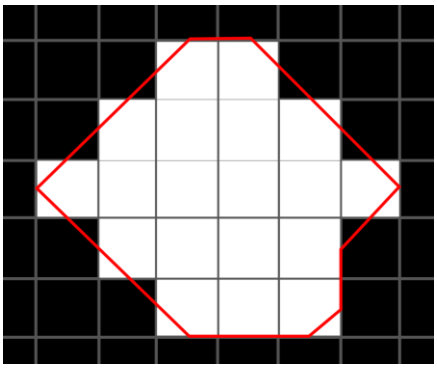

Abbildung 16: Beispiel Marching Square mit stark zu Abbildung 15 abweichender Kontur

Die in Abbildung 16 rekonstruierte Kontur ist stark abweichend von der Originalkontur aus Abbildung 15. Die Kontur konnte nur approximiert werden.

Eine Lösung des Problems ist mit einer Erweiterung des spMC möglich. Die spMC LUT generiert, wie in Abbildung 17 zu sehen, abweichend vom Standard Marching Cube Flächen bzw. Kanten durch die Mittelpunkte der Voxel bzw. Pixel.

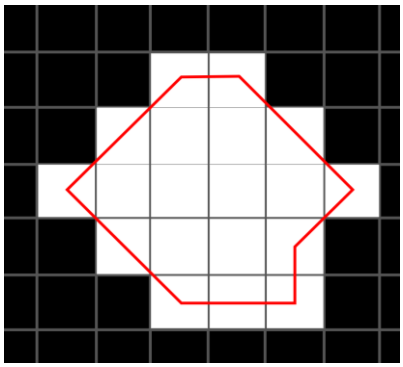

Abbildung 17: Simplified Pattern Marching Square Beispiel als Vorstufe zur Korrektur in Abbildung 18

Die beim Voxelisieren errechneten Schnittpunkte mit den Voxeln können genutzt werden, um die in der Mitte der Voxel erzeugten Dreiecksvertices an die korrekte Position zu verschieben. Man erhält somit ein topologisch korrektes Dreiecksnetz. Abbildung 18 demonstriert dieses Korrekturverfahren anhand des vorherigen Beispiels.

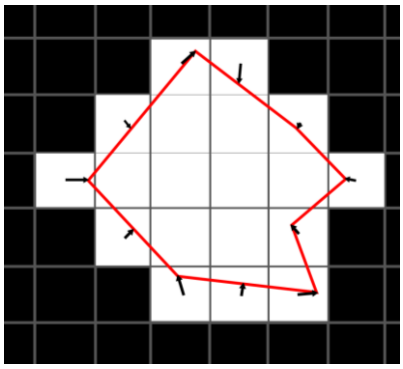

Abbildung 18: Beispiel des Korrekturverfahrens mit rekonstruierter Kontur aus Abbildung 15

Neben der Eigenschaft, dass der spMC Kanten durch die Mittelpunkte der Voxel zeichnet, erzeugt er weniger Dreiecke als der Standard Marching Cube. Durch den kleineren Rechenaufwand ist er zusätzlich schneller.

Inzwischen gibt es zu den weit verbreiteten Derivaten vom Marching Cube vorberechnete LUT. Da der spMC jedoch relativ neu ist und der Veröffentlichung keine LUT beiliegt, wurde diese mit Hilfe von MuPAD/Matlab selbst erstellt. MuPAD ist ein Computeralgebrasystem und Bestandteil der Symbolic Math Toolbox von Matlab. Weiterführendes zur LUT Erstellung ist in Kapitel 3.2 beschrieben.

In Kapitel 2.2.5 wurde beschrieben, dass oft ein Dreiecksnetz Dezimierungsverfahren nach der Triangulation angewendet wird, um eine performantere Weiterverarbeitung zu ermöglichen. Auch in dieser Arbeit wurde die Entscheidung getroffen das Dreiecksnetz nach der Triangulierung zu dezimieren. Für diese Dezimierung der Dreiecke wurde nach weiterem Literaturstudium der in Kapitel 2.2.5 vorgestellte Algorithmus von Schroeder et al. [WJS92] gewählt. In dieser Arbeit wurde ein Dezimierungsverfahren, basierend auf der Veröffentlichung von Schroeder et al., entwickelt und implementiert. Die Eigenentwicklung sollte abweichend kanten- und punkterhaltend sein und außerdem Unterstützung für Multithreading bieten.

Der Ansatz für Multithreading ist angelehnt an den Ansatz von Shontz et al. [Sho13] für eine CPU-GPU Implementierung eines Dezimierungsalgorithmus. Das in dem Algorithmus von Schroeder et al. verwendete Verfahren zur Neutriangulation war für den in dieser Arbeit vorliegenden Anwendungsfall zu aufwändig und unperformant. Deswegen wurde nach einem möglichst einfachen und schnellen Verfahren gesucht. Für die kanten- bzw. punkterhaltende Dezimierung wurde deswegen auf das Half-Edge Collapse Verfahren zurückgegriffen. Der Half Edge Collapse, der in Kapitel 2.2.6 vorgestellt wurde, hat für diesen Anwendungsfall den Vorteil, dass ein Konturpunkt mit einem Nichtkonturpunkt zusammengeführt werden kann, ohne die Position des Konturpunktes zu verändern. Dafür muss auf das Bilden von möglichst gleichgroßen Dreiecken wie beim Full-Edge Collapse oder der rekursiven Schleifentriangulation verzichtet werden. Da jedoch die erzeugten Dreiecke hauptsächlich für die Visualisierung eingesetzt werden, sind langgezogene Dreiecke tolerabel.

Der Marching Cube Algorithmus liefert eine Menge von Dreiecken und keine zusammenhängende Dreiecksnetz-Datenstruktur. Für die Dezimierung und auch für VIRTUOS wird eine Dreiecksnetz-Datenstruktur benötigt um die Beziehungen zwischen den Dreiecken zu erfassen. Außerdem muss jeder Punkt seiner Schicht und Kontur in der VOI Datenstruktur zugewiesen sein. Um ein Dreiecksnetz aufzubauen, wurde die Hierarchical Ring (Abbildung 19) und die Half-Edge (Abbildung 20) Datenstruktur kombiniert [Tri06].

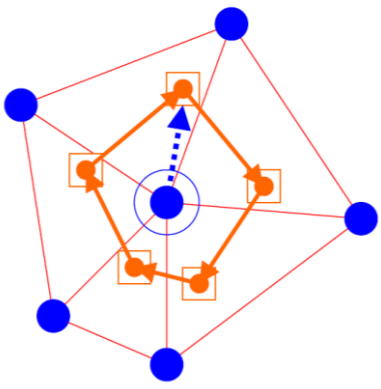

Abbildung 19: Hierarchical Ring [Tri06]

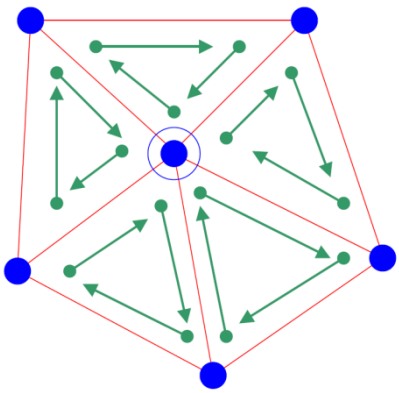

Abbildung 20: Half-Edge Datenstruktur [Tri06]

In der Hierarchical Ring Datenstruktur kennt ein *Vertex* seine benachbarten Dreiecke. Diese werden in einer Liste gespeichert. Die Half-Edge Datenstruktur kennt nur Kanten. Jede Kante zeigt von einem Punkt zu einem Anderen. Drei Kanten bilden hier jeweils ein Dreieck. Die Eigenschaft der Hierarchical Ring Datenstruktur, dass ein *Vertex* seine benachbarten Dreiecke kennt und die Eigenschaft der Half-Edge Datenstruktur, dass Dreiecke nur aus Referenzen zu *Vertices* besteht, sind für diesen Anwendungsfall wünschenswert. Auf dieser Grundlage wurde eine eigene Datenstruktur konzipiert.

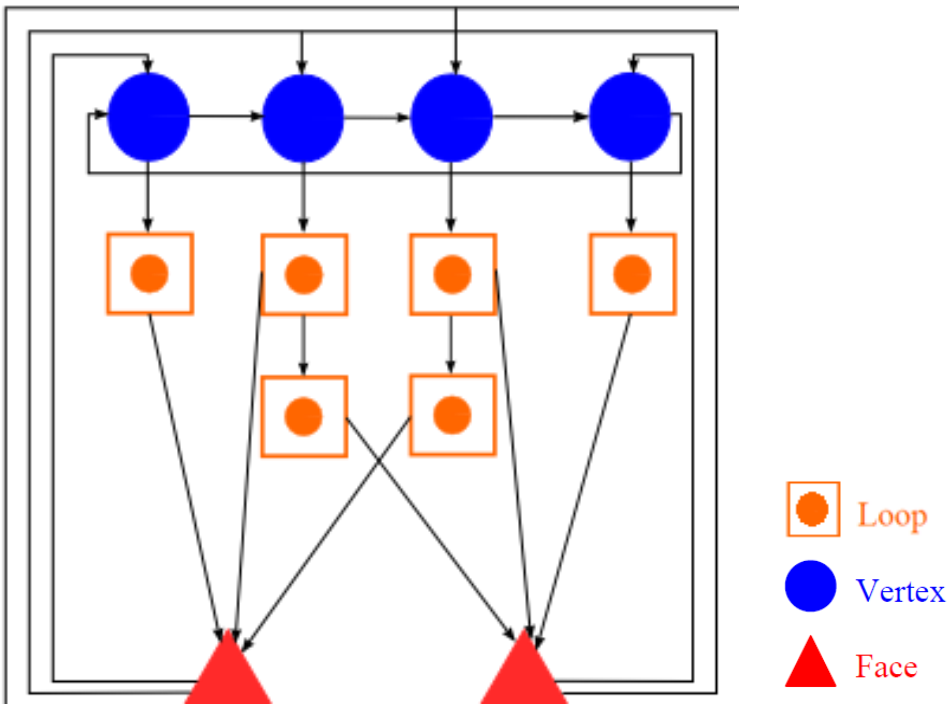

Abbildung 21: Dreiecksnetz-Datenstruktur

Abbildung 21 zeigt die für diese Arbeit verwendete Dreiecksnetz-Datenstruktur. Der *Loop* repräsentiert einen Eintrag in der Liste der benachbarten Dreiecke eines *Vertex*. Ein *Face* repräsentiert ein Dreieck in der Datenstrucktur, das jeweils auf drei Eckpunkte referenziert. Beim Voxelisieren wurden zusätzlich jedem Voxel eine Kontur und eine Schicht zugewiesen. Diese Informationen bleiben bei der Triangulierung erhalten und sind für die VoiManSTL nötig. Details zur Integration sind in Kapitel 3.7 näher erläutert.

Das in diesem Kapitel vorgestellte Verfahren zur Konturerhaltung verhindert nicht, dass ein sogenannter „Treppeneffekt“ in orthogonaler Richtung zu den Konturen entsteht. Dieser resultiert aus vom spMC hinzugefügten Vertices, die nicht auf Konturen liegen. Diese werden somit auch nicht durch das oben beschriebene Verfahren korrigiert. Da es aber keine Einschränkung für diese Punkte gibt, können sie durch ein geeignetes Glättungsverfahren zu einer topologisch genaueren Position korrigiert werden.

Für die Glättung wird in der Literatur meistens der Humphrey's Classes Algorithmus [Bru99] verwendet, da er abweichend von der in Kapitel 2.2.7. vorgestellten Laplace Glättung, das aufgespannte Volumen des Dreiecksnetzes nicht verändert. In diesem Anwendungsfall kann jedoch die einfache Laplace Glättung verwendet werden, wenn man die Punkte, die auf Konturen liegen, von der Glättung ausschließt. Die Punkte, die auf den Konturen liegen, stützen das Dreiecksnetz und verhindern ein „Schrumpfen". Die Verwendung der Laplace Glättung anstatt des Humphrey's Classes Algorithmus bietet durch ihre Einfachheit einen performanz Gewinn.

Der Vorteil, durch die Glättung in Z-Richtung, wird im Folgenden anhand eines Beispiels erläutert.

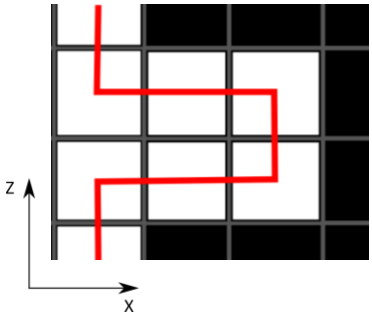

Abbildung 22: ungeglättete Oberfläche in X-Z-Richtung

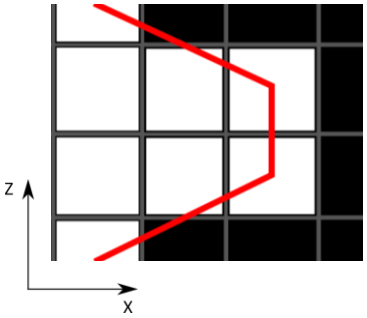

Abbildung 23: geglättete Oberfläche in X-Z-Richtung

Abbildung 22 zeigt durch den spMC verbundene Voxelmittelpunkte in X-Z-Richtung. Die Oberfläche verläuft nicht exakt entlang der durch die Voxelgrenzen definierte Kante, sondern bildet eine Stufe. Punkte, die in X-Richtung nicht am Rand liegen, gehören nicht zu einer Kontur, sondern sind vom Voxelisierer aufgefüllt. Wenn die nicht auf den Konturen liegenden Punkte geglättet werden, erhält man Abbildung 23. Die geglättete Kante ist topologisch genauer als die Ungeglättete.

Details zur Implementierung der vorgestellten Verfahren sind in Kapitel 3.3 zu finden.

3.2 ERSTELLUNG DER LOOKUPTABLE

Wie schon in Kapitel 3.1 erwähnt, war der Veröffentlichung vom spMC [Shi08] keine LUT beigelegt. Diese musste selber erstellt werden.

Im ersten Schritt wurde eine sogenannte EdgeTable erstellt. Diese beinhaltet die Eckvektoren eines Einheitswürfels, der in Abbildung 24 zu sehen ist. Die Dreiecke werden mit diesen Punkten aufgespannt. In der eigentlichen spMC LUT befinden sich nur noch Indices der EdgeTable.

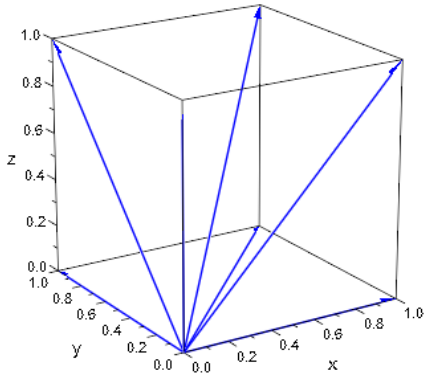

Abbildung 24: Vektoren zu den Ecken des Einheitswürfels

Die EdgeTable folgt dem Schema des ursprünglichen Marching Cube. Dieses fängt bei $(0 \quad 0 \quad 0)^T$ an und geht dann auf unterer Ebene gegen den Uhrzeigersinn. Darauf folgen nach gleichem Schema die Vektoren mit $z = 1$. Es ergibt sich folgende EdgeTable:

$$edgeTable := \left\{\begin{pmatrix}0\\0\\0\end{pmatrix}, \begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}1\\1\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}, \begin{pmatrix}1\\0\\1\end{pmatrix}, \begin{pmatrix}1\\1\\1\end{pmatrix}, \begin{pmatrix}0\\1\\1\end{pmatrix}\right\}$$

Formel 2: Marching Cube EdgeTable

Der spMC setzt seine 254 Verticeskombinationen aus 14 Grund- und 8 Komplementärkonfigurationen zusammen. Alle anderen Konfigurationen sind Drehungen und Spiegelungen der 22 Konfigurationen. Die LUT des spMC ist im Vergleich zum Standard Marching Cube relativ leer, da der spMC Voxelmittelpunkte verbindet und somit mindestens drei markierte Voxel benötigt um eine Dreieck zu bilden. Der Standard Marching Cube erzeugt schon, wie in Abbildung 7 zu sehen, bei einem markierten Voxel in einer Konfiguration ein Dreieck. In Abbildung 25 sind alle Grund- und Komplimentärkonfigurationen des spMC aufgeführt. Die Komplementärkonfigurationen sind anhand des angehangenen „c“ in der jeweiligen Beschriftung erkenntlich gemacht.

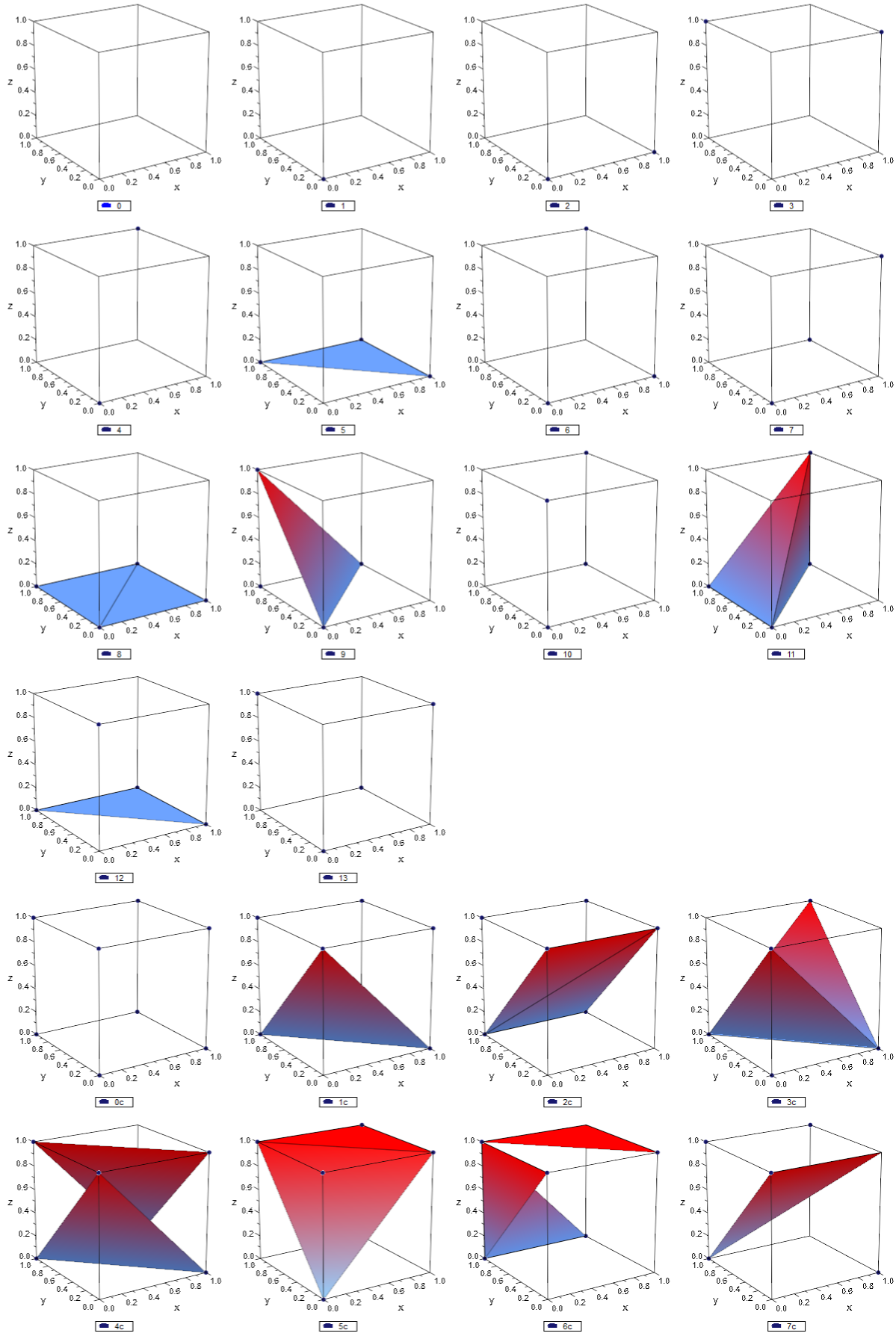

Abbildung 25: spMC Grund- und Komplementärkonfigurationen

Die komplette spMC LUT wurde erstellt, indem die 22 Konfigurationen manuell eingetragen wurden und die Anderen durch Kombination von Translation in den Ursprung, Rotation/Spiegelung mittels der folgenden Matrizen und Rücktranslation an die ursprüngliche Position errechnet und eingetragen wurden. Die Spiegelungsmatrizen für die Y- und Z-Achse können mittels Drehung der Spiegelungsmatrix für die X-Achse ermittelt werden.

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$S_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Formel 3: Rotationsmatrizen und Spiegelungsmatrix der X-Achse

Danach wurden die Punkte in Indices der EdgeTable umgerechnet und anhand dieser die Position der Konfiguration in der LUT ermittelt. Dort wurden die Punktindices eingetragen.

Damit die Richtung des Normalenvektors der Dreiecke implizit gegeben ist, wurden die Vertices in der LUT gegen den Uhrzeigersinn angeordnet. Dies bietet die Möglichkeit den Normalenvektor des Dreiecks wie folgt zu berechnen.

$$\vec{n} = (\vec{v}_1 - \vec{v}_2) \times (\vec{v}_1 - \vec{v}_3)$$

Formel 4: Normalenvektor auf einem Dreieck

Die generierte spMC Lookuptable kann dem Kapitel 8.3 im Anhang entnommen werden.

3.3 Ablauf der Triangulierung

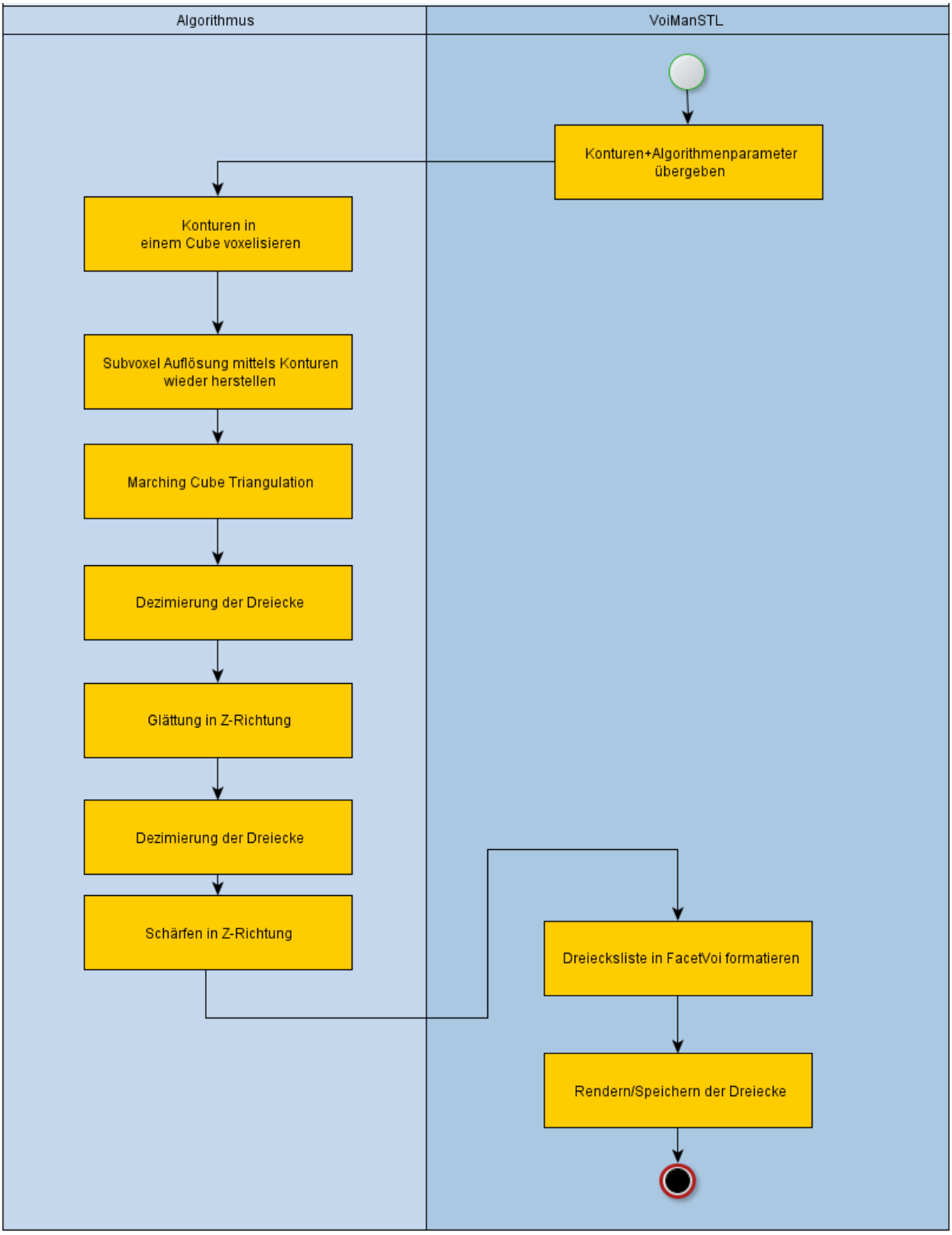

Abbildung 26: UML Ablaufdiagramm vom Algorithmus

Die Abbildung 26 zeigt die entworfene Pipeline von Teil-Algorithmen, die der im Rahmen dieser Arbeit entwickelte Algorithmus durchläuft. Im Folgenden werden die einzelnen Schritte kurz erklärt.

1. Wenn die VoiManSTL von VIRTUOS das hier entwickelte Modul aufruft, muss es die Konturen und Algorithmen-Parameter angeben. Die Parameter beeinflussen das Verhalten des Algorithmus und erfüllen die in Kapitel 1.3.2 beschriebene Anforderung für eine umfangreiche Parametrisierung des Algorithmus.
2. Anhand der Konturen wird ein Volumen im Voxelraum markiert und gefüllt. Man erhält ein 3D-Binärbild.
3. Die Punkte hinter den markierten Voxeln werden jetzt, anhand der Schnittpunkte mit den Voxeln, nach dem in Kapitel 3.1 vorgestellten Verfahren, korrigiert um den typischen Marching Cube „Treppeneffekt“ in X-Y-Richtung zu entfernen und die Auflösung zu verbessern.
4. Mittels dieses Volumens kann nun die Oberfläche mit Hilfe des Simplified Pattern Marching Cube Algorithmus ermittelt werden. Die Oberfläche wird dabei in Millimeter-Koordinaten generiert.
5. Damit die Oberfläche flüssig gerendert werden kann, werden die durch den Marching Cube erstellten Dreiecke dezimiert.
6. Um auch eine Glättung in Z-Richtung zu erreichen, werden Punkte, die nicht auf Konturen liegen, mittels der Laplace Dreiecksnetz Glättung geglättet.
7. Da nach der Glättung wieder Dreiecke dezimierbar sind, wird nun nochmal eine weitere Dreiecksdezimierung durchgeführt.
8. Durch die Glättung können Punkte zwischen zwei Schichten liegen. Da das VoiManSTL Datenformat nur Punkte auf Schichten erlaubt, werden diese in diesem Schritt an ihre ursprüngliche Position vor der Glättung verschoben.
9. Das Dreiecksnetz wird an die VoiManSTL übergeben. Nach einer Formatierung von der eigenen Datenrepräsentation in das *FacetVoi* Format kann die triangulierte Fläche abgespeichert oder angezeigt werden.

3.4 DESIGN

Beim Architekturdesign wurde eine schlanke Schnittstelle angestrebt. Alle Funktionen sind über die Schnittstelle der Hauptklasse *MCCPU* ansprechbar. Die Architektur wurde mit der Unified Modeling Language (UML) modelliert.

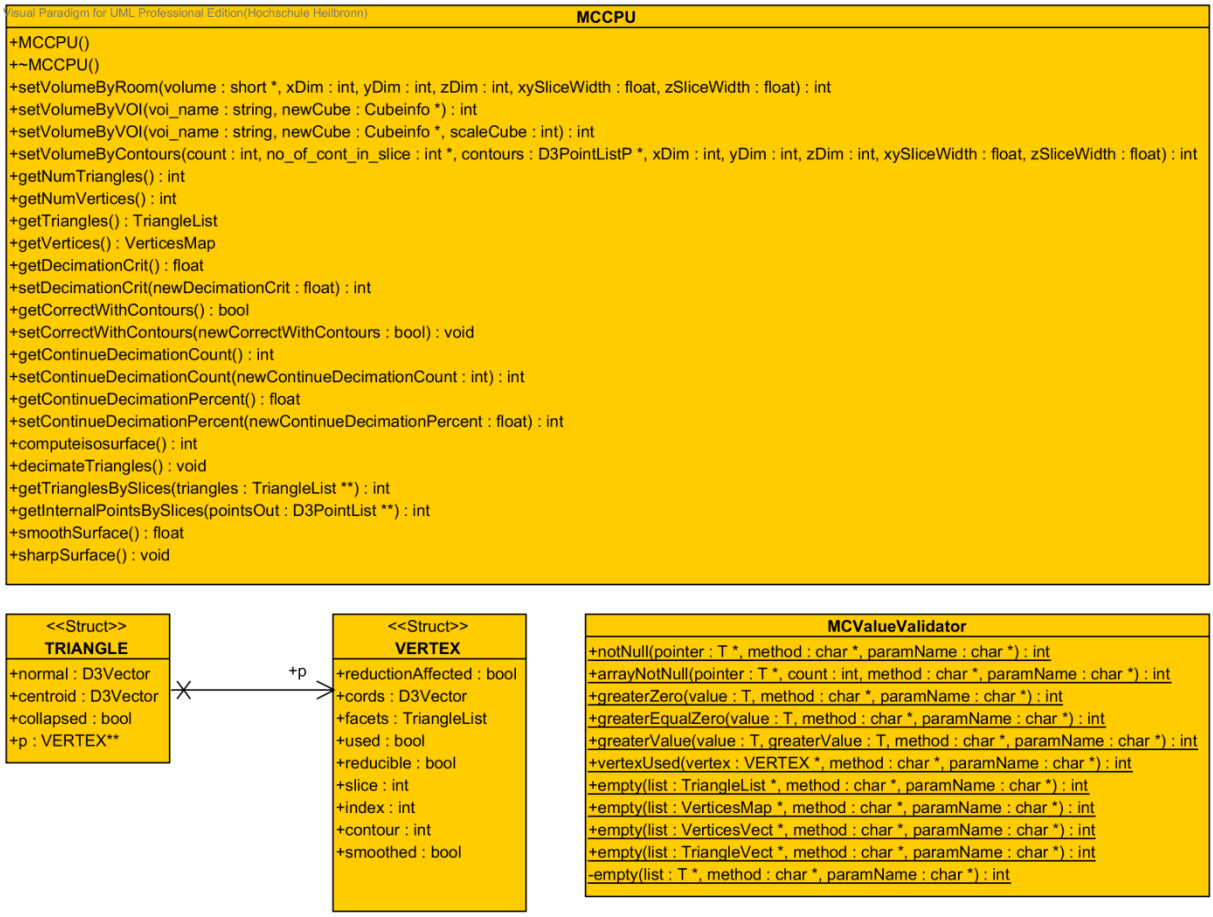

Abbildung 27: Klassendiagramm

Neben der Hauptklasse *MCCPU* gibt es die Hilfsklasse *MCValueValidator* in der Variablenprüfungen ausgelagert sind. Des Weiteren gibt es die zwei Strukturen *TRIANGLE* und *VERTEX* zur Repräsentation von Dreiecken bzw. Vertices im Algorithmus.

Das Interface der Klasse *MCCPU* bietet Setter zum manuellen Setzen eines Volumens anhand eines 3D Skalarfeld, zum Setzen eines Volumens anhand eines VOI Namens, des aktuell geladenen VOI Modells und zum Setzen eines Volumens anhand eines Konturstapels. Die Korrektur der Punkte kann mittels des Datenfeldes *correctWithContours* an- bzw. abgeschaltet werden.

Die Triangulierung kann mittels *computeIsosurface()* angestoßen werden.

Die Methode decimateTriangles() dezimiert die erstellten Dreiecke. Mit den Datenfeldern *continueDecimationCount*, *continueDecimationPercent*, *decimationCrit* kann die Dezimierung beeinflusst werden. Wenn in einer Dezimierungsiteration weniger als im Datenfeld *continueDecimationCount* festgelegte Dreiecke dezimiert wurden, bricht die Dezimierung ab. Zusätzlich bricht die Dezimierung ab, wenn nur noch weniger als *continueDecimationPercent* Prozent Dreiecke übrig sind. Das Datenfeld *decimationCrit* beeinflusst die zweite Dezimierung. Hier werden Punkte auf internen Konturen dezimiert. Dieses Datenfeld legt fest, mit welchem Abstand, in CT-Koordinaten, zu einer Fläche bzw. einer Geraden ein Punkt dezimiert wird. Dieses Datenfeld entspricht dem Abstand **d,** der in Kapitel 2.2.5 erwähnt wurde. Wenn *decimationCrit* größer Null ist, erfolgt vor der zweiten Dezimierung ein Glätten der Oberfläche und nach der Dezimierung ein Schärfen der Oberfläche.

Mittels der Funktionen *smoothSurface()* bzw. *sharpSurface()* kann das Glätten bzw. Schärfen der Oberfläche manuell ausgelöst werden.

Die Getter-Methoden *getNumTriangles()*, *getNumVertices()*, *getTriangles()*, *getVertices()*, *getTrianglesBySlices()* und *getInternalPointsBySlices()* bieten die Möglichkeit das Resultat der Triangulierung zu bekommen.

3.5 IMPLEMENTIERUNGSDETAILS

In diesem Abschnitt wird auf ausgewählte relevante Details der Implementierung des Algorithmus eingegangen. Dabei wird außerdem auf die Generierung der speziellen LUT und die Anbindung an Drittsoftware zu Testzwecken eingegangen.

3.5.1 VOXELISIERUNG

Die Voxelisierung erfolgt mittels VIRTUOS Funktionen. Dafür werden die übergebenen Konturen in einen VIRTUOS VOI Würfel (cube) voxelisiert. Vor diesem Vorgang müssen die Konturen in die Cube-Koordinaten umgerechnet werden.

Das Voxelisieren nutzt die VIRTUOS Funktion *d3_fill_area* um alle Voxel innerhalb der Konturen zu markieren. Nur angeschnittene Voxel werden mit der Funktion *d3_fill_3d_line* markiert.

Die Funktion *d3_fill_3d_line* liefert eine Liste der Schnittpunkte von Voxeln und Konturen zurück. Diese wird im Folgenden genutzt um die in Kapitel 3.1 beschriebene Korrektur durchzuführen. Alle Vertices werden in einer HashMap gespeichert. Der Hashwert jedes Eintrags bietet die Möglichkeit nachzuvollziehen, in welchem Voxel sich der jeweilige Punkt befindet. Nun wird jeder Punkt der Liste unter seinem jeweiligen Hashwert eingetragen. Bei einem Würfel mit den Dimensionen $cube_{\mathrm{x}} \times cube_{\mathrm{y}} \times cube_{\mathrm{z}}$ ist der Hashwert für einen Punkt i in Würfel-Koordinaten wie folgt definiert.

$$index_i = \lfloor x_i \rfloor + \lfloor y_i \rfloor \cdot cube_x + \lfloor z_i \rfloor \cdot cube_x \cdot cube_y$$

Formel 5: Hashwert eines Punktes

Des Weiteren wird hinter den Punkten die jeweilige Kontur, Schicht und ob sie sich an einer Ecke oder innerhalb einer Linie auf der Kontur befinden, hinterlegt.

3.5.2 TRIANGULIERUNG

Die Triangulierung mittels des spMC weicht nicht stark vom Original [Shi08] ab. Die nennenswerteste Abänderung ist das Erstellen der in Kapitel 3.1 beschriebenen Datenstruktur.

Wenn neue Punkte in die Datenstruktur eingefügt werden müssen, werden diese der jeweiligen internen Kontur der Schicht zugewiesen. Eine interne Kontur beinhaltet alle Punkte, die nicht auf anderen Konturen liegen. Sie ist durch den Benutzer nicht direkt veränderbar, bietet aber die Möglichkeit, Punkte außerhalb der vom Nutzer vorgegebenen Konturen zu definieren. Während der Triangulierung werden zusätzlich die Normale und der geometrische Schwerpunkt (Centroid) der erzeugten Dreiecke errechnet.

Die Triangulierung kann für jeden logischen Würfel theoretisch gleichzeitig durchgeführt werden. Deswegen wurde die Triangulierung mittels OpenMP parallelisiert.

3.5.3 DREIECKSNETZ DEZIMIERUNG

Der in Kapitel 2.2.5 beschriebene Dreiecksnetz-Dezimierungs-Algorithmus wurde für die Parallelisieren mit OpenMP modifiziert. Damit die unterschiedlichen Threads unabhängig laufen können, muss das Problem in disjunkte Teilprobleme geteilt werden. Abbildung 28 zeigt verschiedenfarbig zwei disjunkte Regionen auf dem Dreiecksnetz. Die in der Abbildung hervorgehobenen Vertices können unabhängig dezimiert werden, da sich die zwei Regionen kein gemeinsames Dreieck teilen. Dadurch können zwei Threads Dreiecke in ihrer jeweiligen Region kollabieren lassen, ohne von den Dreiecksnetzänderungen des anderen Threads abhängig zu sein. Um eine gleichmäßigere Dezimierung zu erhalten, wird die Abarbeitungsreihenfolge der Kandidaten vor dem Aufteilen der Regionen gemischt. Dadurch verhindert man, dass die Oberfläche immer ähnlich in einzelne Regionen aufgeteilt wird.

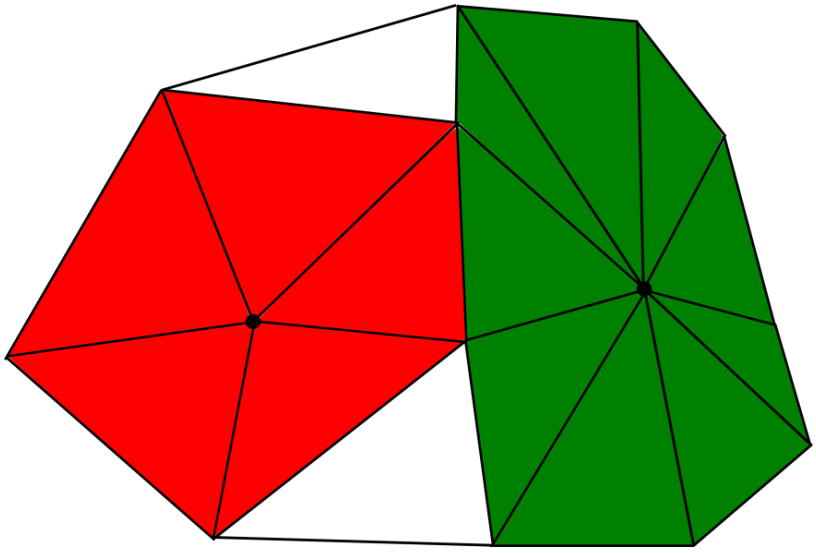

Abbildung 28: Disjunkte Regionen im Dreiecksnetz

Dafür werden die einzelnen Regionen sequenziell markiert und die Kandidaten für die Dezimierung in einer Liste gespeichert. Vertices, die Konturreckpunkte repräsentieren, werden ausgenommen, da sie nicht dezimiert werden dürfen. Das in Kapitel 2.2.5 beschriebene Verfahren zum Kategorisieren der Vertices wurde in zwei unterschiedlichen Abwandlungen implementiert.

Die schnellere Variante versucht „flache“ Simple- und Interior Edge Vertices zu dezimieren. Ein Vertex wird als Simple Vertex markiert, wenn alle Normalenvektoren n_{i} der benachbarten Dreiecke folgende Gleichung erfüllen.

$$n_1 \cdot n_i = 0$$

Formel 6: einfaches Simple Vertex Kriterum

Die Normalenvektoren der Nachbardreiecke zeigen somit alle in die gleiche Richtung. Mit einem erkannten Simple Vertex wird versucht ihn, mittels dem in Kapitel 3 vorgestelltem Half-Edge Collapse, mit seinen euklidisch nächsten Nachbarn zu dezimieren. Interior Edge Vertices haben Nachbardreiecke, bei denen ein Teil der Normalenvektoren in eine, ein anderer Teil in eine zweite Richtung zeigt. Um dies zu erkennen, wird wieder Formel 6 verwendet. Zu einem erkannten Interior Edge Vertex wird versucht, mittels der Lösung einer Geradengleichung, eine Linie durch zwei Nachbarn und dem Vertex selbst zu ziehen. Wenn dies gelingt, wird versucht den Vertex mit dem in Kapitel 3.1 vorgestelltem Half-Edge Collapse an einen der zwei Nachbarn zu dezimieren.

Die langsamere Variante der Kategorisierung der Vertices funktioniert analog zur schnellen Variante. Jedoch werden nur Punkte auf internen Konturen dezimiert. Außerdem werden zum Kategorisieren die wie in Kapitel 2.2.5 beschriebenen Metriken verwendet.

In bestimmten Fällen kann die Dezimierung mit dem Half-Edge Collapse fehlschlagen. Deswegen wird der Zustand der zusammenzuführenden Vertices und der benachbarten Dreiecke auf einem Stack gespeichert. Um einen Fehlschlag zu erkennen, wird überprüft, wie viele Dreiecke kollabieren. In Kapitel 2.2.6 wurde beschrieben, dass beim Edge Collapse nur zwei Dreiecke kollabieren dürfen, damit ein topologisch korrektes Dreiecksnetz erhalten bleibt. Außerdem wird anhand der Normalenvektoren überprüft, ob sich die Orientierung der Nachbardreiecke geändert hat. Wenn sich ein Dreieck umgedreht hat oder zu viele Dreiecke kollabiert sind, wird der Vorgang, mittels des Stacks, rückgängig gemacht. Der gleiche Mechanismus wird außerdem genutzt um zu verhindern, dass bei der Dezimierung lange Dreiecke entstehen. Dafür werden als zusätzliche Überprüfung die Längen der Dreiecksseiten nach dem Half-Edge Collapse verglichen und gegebenenfalls der Half-Edge Collapse rückgängig gemacht.

3.5.4 OBERFLÄCHEN GLÄTTUNG UND SCHÄRFUNG

Nach einem „Ausdünnen“ durch die schnelle Dezimierung folgt ein Glätten aller Vertices, die sich auf internen Konturen befinden. Dafür wird die in Kapitel 2.2.7 beschriebene Laplace Glättung verwendet.

Um die Geschwindigkeit zu erhöhen und Ungenauigkeiten durch Rundungsfehler zu vermeiden wird überprüft, ob ein Kandidat zu einem „flachen“ Simple Vertex gehört. Dafür wird das Verfahren aus Kapitel 3.5.3 verwendet. „Flache“ Simple Vertices werden von der Glättung ausgeschlossen. Die Abarbeitungsreihenfolge der verbliebenden Kandidaten wird, um eine gleichmäßigere Glättung zu erhalten, gemischt.

Die Punkte werden nun mittels der Laplace Glättung geglättet.

Bei diesem Verfahren können Dreiecke, wie beim Edge Collapse, kollabieren oder sich umdrehen. Um Vertex Verschiebungen rückgängig machen zu können, wird das gleiche Verfahren wie das für den Edge Collapse aus Kapitel 3.5.3 verwendet.

Je nachdem, wie die Oberfläche aufgebaut ist, muss der Glättungsvorgang mehrfach durchgeführt werden. Es hat sich als erfolgreicher erwiesen, ein Abbruchkriterium anhand einer Metrik zu verwenden, anstatt die Anzahl der Iterationen statisch festzulegen. Die hier verwendete Metrik summiert die Beträge der Verschiebungsvektoren aller Punkte im Dreiecksnetz. Wenn dieser Wert für eine Iteration kleiner als ein Schwellwert ist, wird keine weitere Glättung durchgeführt. Der verwendete Schwellwert wurde experimentell ermittelt.

Das Datenmodell der VoiManSTL gibt vor, dass sich Punkte nur auf den vorgegebenen Schichten befinden dürfen. Durch die Glättung konnten sich Punkte unabhängig bewegen. Um ein Ergebnis zu erhalten, das ähnlich dem mit Glättung ist, sich die Punkte aber nur auf Schichten befinden, wurde eine Kombination aus Glättung und Dezimierung verwendet. Dafür wurde die langsamere Dezimierung aus Kapitel 3.5.3 nach der Glättung angewandt, um die geglätteten Punkte zu dezimieren. Es kann vorkommen, dass nicht alle geglätteten Punkte dezimiert werden können. Deswegen werden diese nach der Dezimierung wieder an die ursprüngliche Position vor der Glättung geschoben. Dieses Vorgehen wird im Rahmen dieser Arbeit als „Schärfen“ bezeichnet.

Die Position in Würfel-Koordinaten kann mit folgenden Formeln aus dem mit Formel 5 errechneten Hashwert $index_{\mathrm{i}}$ eines Vertex zurückgerechnet werden. Dabei hat der Würfel die Dimensionen $cube_{\mathrm{x}} \times cube_{\mathrm{y}} \times cube_{\mathrm{z}}$

$$x_i = \left(index_i \; mod \left(cube_{\mathrm{x}} \cdot cube_{\mathrm{y}}\right)\right) mod \; cube_{\mathrm{x}}$$

$$y_i = \frac{(index_i - x_i) \; mod \left(cube_{\mathrm{x}} \cdot cube_{\mathrm{y}}\right)}{cube_{\mathrm{x}}}$$

$$z_i = \frac{index_i - x_i - y_i \cdot cube_{\mathrm{x}}}{cube_{\mathrm{x}} \cdot cube_{\mathrm{y}}}$$

Formel 7: Vertex Position aus Hashwert errechnen

3.6 ANBINDUNG AN MUPAD/PARAVIEW

Um bessere Möglichkeiten zu bieten den Algorithmus testen zu können, wurde eine Schnittstelle zu MuPAD/Matlab erstellt. MuPAD ist Teil vom Computeralgebra Programm Matlab. Es ist leicht zu verwenden, bietet aber nicht den kompletten Funktionsumfang von Matlab.

Mit der Schnittstelle zu MuPAD ist es möglich Volumen in MuPAD zu generieren und zu exportieren. Außerdem ist es möglich die vom Algorithmus generierten Dreiecke zu importieren, darzustellen und zu evaluieren.

Um Volumen aus MuPAD zu exportieren, wurde ein einfaches Dateiformat entworfen, das möglichst einfach durch C/C++ verarbeitbar ist. Es beinhaltet einen Dateikopf mit den Dimensionen des Volumens, getrennt durch jeweils einem Zeilenumbruch. Darauf folgen alle markierten Voxel, jeweils beschrieben mit ihren Koordinaten im Voxelraum, getrennt durch Kommata. Die Voxel sind wieder mit einem Zeilenumbruch getrennt. Das folgende Beispiel zeigt eine einfache Datei mit den Dimensionen 10x10x10 und vier markierten Voxeln.

```
10
10
10
5, 5, 5
6, 5, 5
7, 5, 5
8, 5, 5
```

Abbildung 29: Listening MuPAD Export Beispieldatei

Zum Import der Dreiecke in MuPAD wurde wiederum in C/C++ eine MuPAD Source Code Datei geschrieben. Diese beinhaltet die Definition einer Variablen „triangleList“, die alle erzeugten Dreiecke samt ihrer Normalenvektoren enthält. Weiterführendes zur Formatierung von Dreieckslisten in MuPAD ist im offiziellen MuPAD Handbuch zu finden [Sym08].

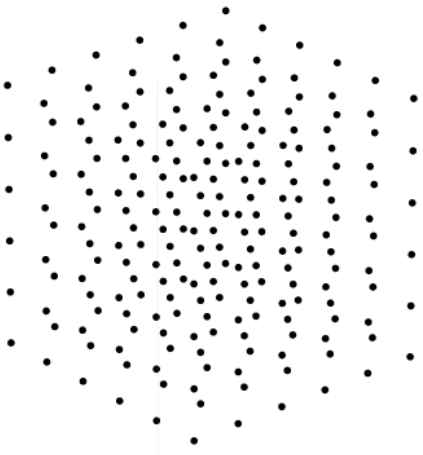

Abbildung 30: Voxelraum in MuPAD

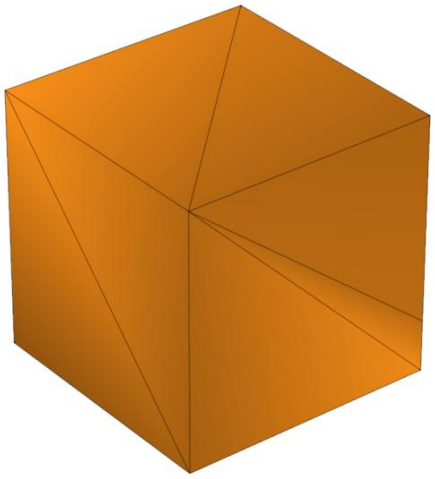

Abbildung 31: Importierte Dreiecke in MuPAD

Beim Testen der Schnittstelle ist aufgefallen, dass die Renderzeit von MuPAD ab einer Anzahl von ca. 40000 Dreiecken enorm ansteigt. Viele der verwendeten Beispiel VOIs haben nach der Triangulierung jedoch deutlich mehr Dreiecke. Deswegen wurde eine Anbindung an ParaView realisiert. ParaView ist eine auf dem Visualization Toolkit (VTK) basierende Open-Source-Software für wissenschaftliche 3D Visualisierung.

Um eine Schnittstelle mit ParaView zu schaffen, wurde der Export des Dreiecksnetz über das Surface Tesselation Language (STL) [3DS89] Format implementiert. Für eine einfachere Implementierung wurde das ASCII STL Format gewählt

Die ASCII Version des STL-Formats besteht aus Blöcken, die jeweils Kindknoten enthalten. Der Wurzelknoten definiert den Namen der 3D Struktur. Er beinhaltet Blöcke von Polygonschleifen mit den jeweiligen Normalenvektoren. Diese wiederum beinhalten die Eckpunkte der jeweiligen Schleife. Das nachfolgende Beispiel zeigt eine einfache STL-Datei mit einem Dreieck.

```
solid BEISPIELOBJEKT
facet normal 0 0 1.0
  outer loop
    vertex 0 0 0
    vertex 0 1.0 0
    vertex 1.0 0 0
  endloop
endfacet
endsolid BEISPIELOBJEKT
```

Abbildung 32: Listening ASCII STL Beispieldatei

Abbildung 33: 257167 Importierte Dreiecke in ParaView

3.7 VIRTUOS INTEGRATION

Das bisherige Triangulierungsmodul, das die Delaunay Triangulation verwendet, ist stark in die VoiManSTL eingebunden. Dadurch war es schwer eine lose Kopplung für das in dieser Arbeit erstellte Modul zu erreichen. Somit wurde eine ähnliche Integration angestrebt. Diese ruft aus der VoiManSTL das in dieser Arbeit entwickelte Modul auf. Abbildung 34 zeigt die Aufrufkaskade vom Einsprungpunkt, in der VIRTUOS GUI, bis zum Marching Cube Modul.

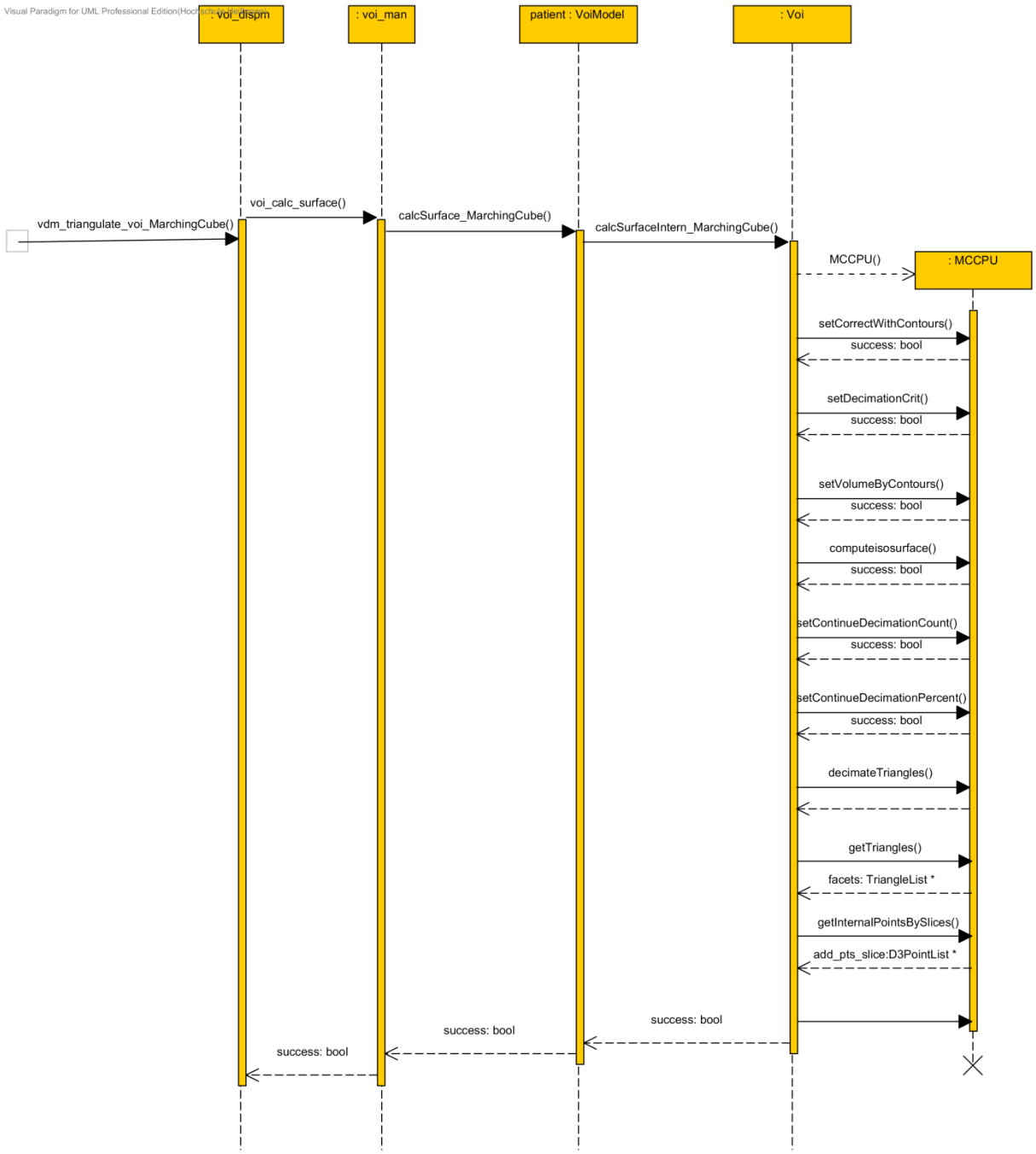

Abbildung 34: Sequenzdiagramm vom Aufruf des Marching Cube Moduls

Neben dem, im Rahmen dieser Arbeit entwickelten Modul, steckt die für die Triangulierung wichtigste Logik in der Funktion *calcSurfaceIntern_MarchingCube()*. Diese Funktion legt eine Sicherung der Konturen an, initialisiert alle für die Triangulierung nötigen Datenstrukturen und stellt sie bereit. Danach startet sie die Triangulierung, pflegt die errechneten internen Konturpunkte in die ContourVoi Datenstruktur ein und erstellt eine neue *FacetVoi*, die mit den errechneten Dreiecken gefüllt wird. Abbildung 35 zeigt den beschriebenen Ablauf der *calcSurfaceIntern_MarchingCube()* Funktion im Detail.

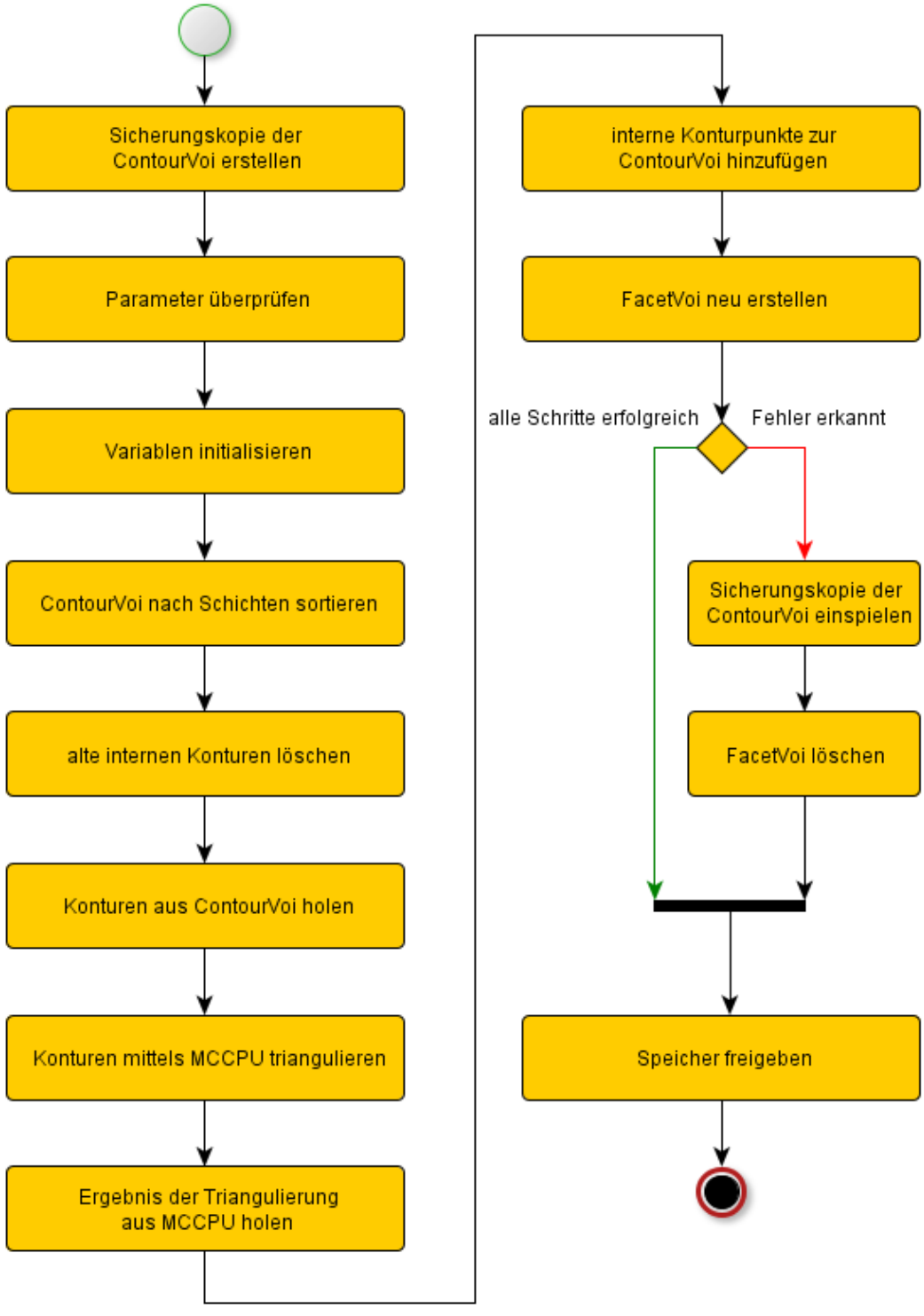

Abbildung 35: Ablauf der Funktion *calcSurfaceIntern_MarchingCube()*

3.8 TRIANGULATIONSERGEBNISSE

In diesem Kapitel wird anhand verschiedener Beispielbilder auf Eigenschaften des im Rahmen dieser Arbeit entwickelten Algorithmus eingegangen. Die gezeigten Bilder wurden in VIRTUOS erstellt und mittels Screenshots aufgenommen.

Abbildung 36 zeigt eine Kante eines mit dem spMC triangulierten Quader von oben. Zum Vergleich zeigt Abbildung 37 den gleichen Quader mit der im Rahmen dieser Arbeit entwickelten Korrekturmethode aus Kapitel 3.1, die eine Auflösung im Subpixelbereich ermöglicht.

Abbildung 36: triangulierter Quader ohne Korrektur

Abbildung 37: triangulierter Quader mit Korrektur

Abbildung 38 zeigt einen linken Lungenflügel, welcher mit 107229 Dreiecken dargestellt wird. Abbildung 39 zeigt im Vergleich den gleichen Lungenflügel nach der Dreiecksnetzdezimierung mit 67086 Dreiecken. Trotz starker Dezimierung fallen nur im Detail Unterschiede auf.

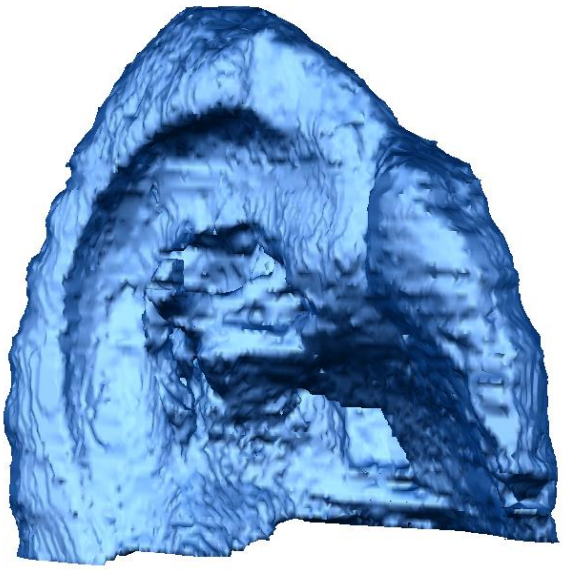

Abbildung 38: triangulierter Lungenflügel vor der Dezimierung mit 107229 Dreiecken

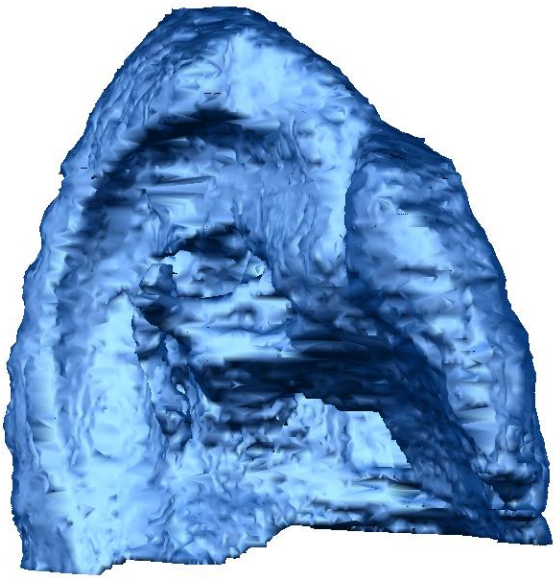

Abbildung 39: triangulierter Lungenflügel nach der Dezimierung mit 67086 Dreiecken

Abbildung 40 zeigt ein Herz ohne Glättung der internen Konturpunkte. Abbildung 41 zeigt das Ergebnis nach der Glättung. Die Glättung ist in Z-Richtung besonders sichtbar.

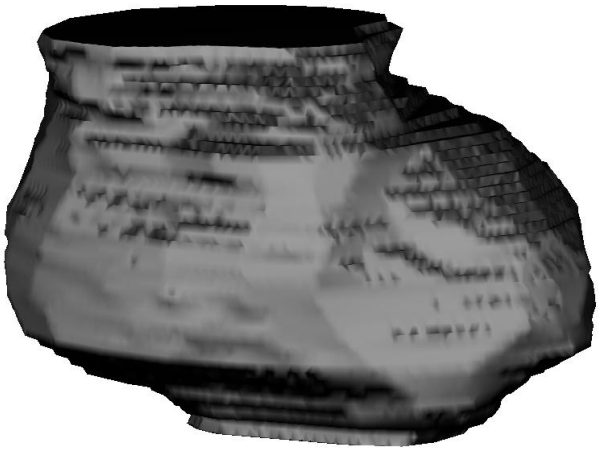

Abbildung 40: trianguliertes Herz ohne Glättung

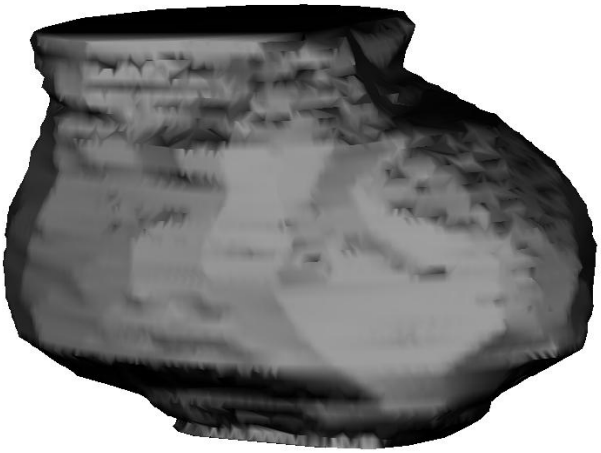

Abbildung 41: trianguliertes Herz mit Glättung

Abbildung 42, Abbildung 43 und Abbildung 44 zeigen das Ergebnis der Triangulation einer Patientenoberfläche mittels der bisherigen Delaunay Triangulation, der neuen Triangulation und der neuen Triangulation mit kleiner gewählten Voxelrasterung. Für die gröbere Voxelrasterung wurden jeweils 9 Voxel des CT-Datensatzes zu einem zusammengefasst. Dies ermöglicht eine deutlich schnellere Triangulierung. Ein Performanz Vergleich der Algorithmen ist in Kapitel 3.9 zu finden.

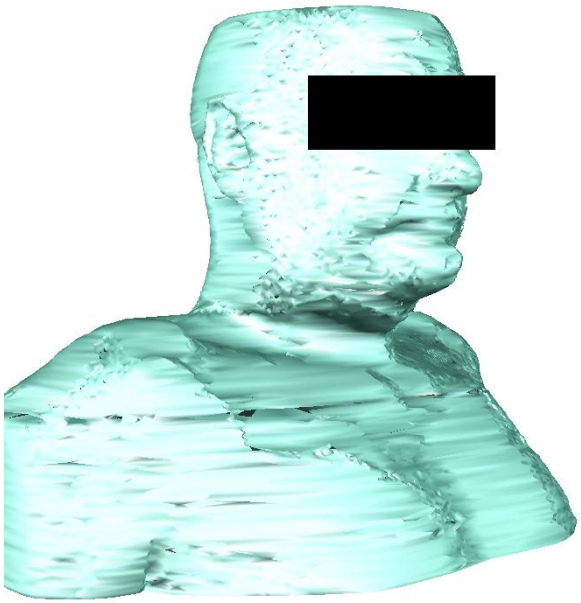

Abbildung 42: mit der Delaunay Triangulation triangulierte Patientenoberfläche

Abbildung 43: mit dem im Rahmen dieser Arbeit entwickelten Algorithmus triangulierte Patientenoberfläche

Abbildung 44: mit dem im Rahmen dieser Arbeit entwickelten Algorithmus triangulierte Patientenoberfläche mit gröberem Voxelraster

Abbildung 45, Abbildung 46 und Abbildung 47 zeigen wiederum das Ergebnis der Triangulation eines Rückenmarks mittels der bisherigen Delaunay Triangulation, der neuen Triangulation und der neuen Triangulation mit kleiner gewählten Voxelrasterung. Eine Schwäche aller Triangulierungsalgorithmen ist das Verbinden von fast bis komplett disjunkten Konturen. Besonders der neue Algorithmus bildet prinzipbedingt nur einen dünnen Steg zwischen den Konturen.

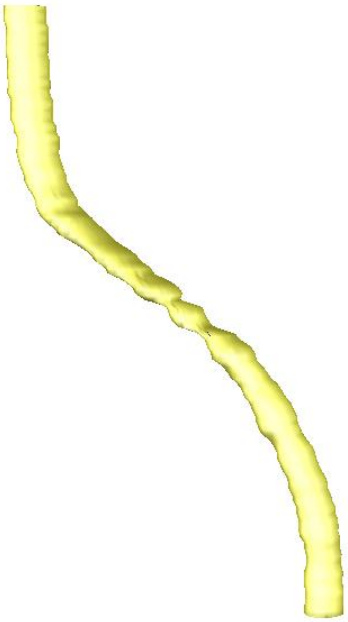

Abbildung 45: mit der Delaunay Triangulation trianguliertes Rückenmark

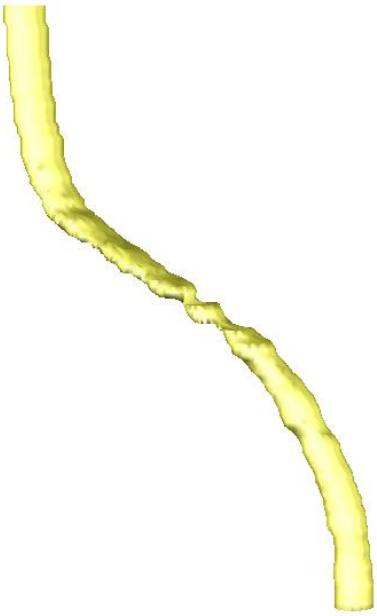

Abbildung 46: mit dem im Rahmen dieser Arbeit entwickelten Algorithmus trianguliertes Rückenmark

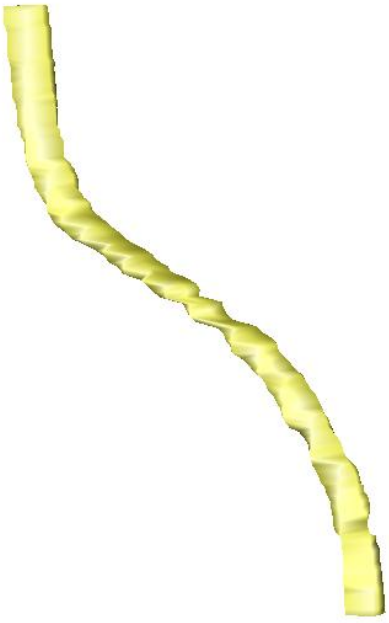

Abbildung 47: mit dem im Rahmen dieser Arbeit entwickelten Algorithmus trianguliertes Rückenmark

Eine für Triangulationsalgorithmen wichtige Eigenschaft ist das Verhalten beim Triangulieren von sich überschneidenden Konturen (intersections). Das Verhalten, von dem im Rahmen dieser Arbeit entwickelten Algorithmus, beim Triangulieren von sich überschneidenden Konturen, hängt maßgeblich vom Verhalten des verwendeten Voxelisier-Algorithmus ab. Der in dieser Arbeit verwendete Voxelisierungs-Algorithmus ist schon bestehender Bestandteil von VIRTUOS. Die folgenden Beispiele zeigen, wie sich diese Voxelisierungs-Strategie auf das Triangulationsergebnis auswirkt.

Abbildung 48 zeigt eine VOI mit mehreren sich selbst überschneidenden Konturen. Abbildung 49 zeigt das zugehörige Triangulationsergebnis. Um das komplette eingeschlossene Volumen wurde eine Oberfläche gezogen.

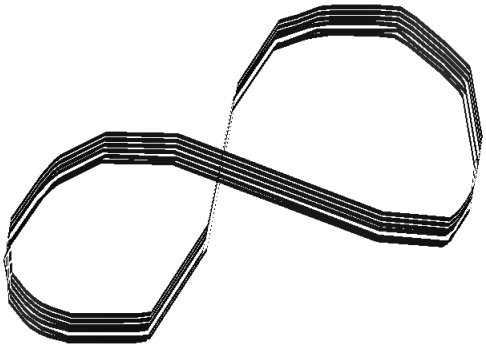

Abbildung 48: überschneidene Konturen als Bändermodell

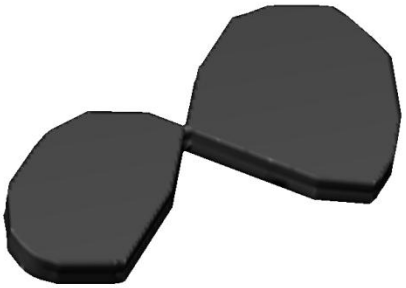

Abbildung 49: triangulierter Konturstapel aus Abbildung 48 mit sich überschneidenen Konturen

Abbildung 50 zeigt mehrere Schichten mit jeweils drei ineinander verschachtelten Konturen. Abbildung 51 zeigt das zugehörige Triangulationsergebnis. Die sich überlagerten Flächen wurden durch den Voxelisierer nicht markiert und wurden somit beim Erzeugen der Oberfläche ausgespart.

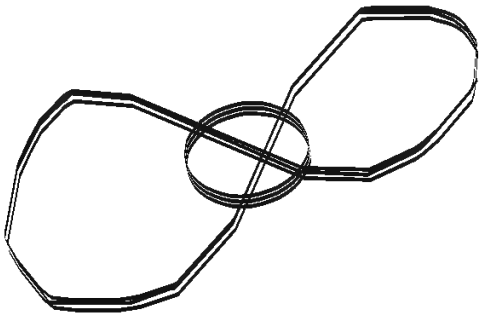

Abbildung 50: verschachtelte Konturen als Bändermodell

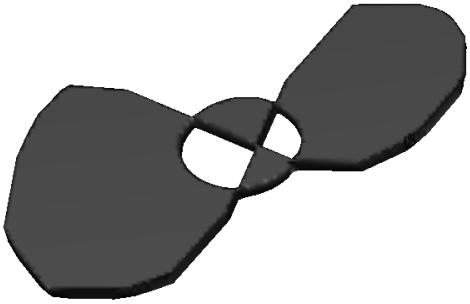

Abbildung 51: triangulierter Konturstapel aus Abbildung 50 mit verschachtelten Konturen

3.9 PERFORMANZ

Dieses Kapitel soll einen Überblick über die Performanz des im Rahmen dieser Arbeit entwickelten Algorithmus geben. Das Kapitel behandelt die Geschwindigkeitssteigerung durch Parallelisierung mit OpenMP. Es wird auf die Auswirkung der Parametrisierung auf die Performanz und die erzeugten Dreiecke eingegangen. Außerdem wird der Algorithmus mit der schon vorhandenen Delaunay Triangulation verglichen.

Ein Performanz Vergleich der hier verglichenen Algorithmen ist abhängig von der verwendeten Hardware. Tabellen mit den für die Graphen verwendeten Werten und der Algorithmen Parametrisierung sind im Anhang unter 8.2 zu finden.

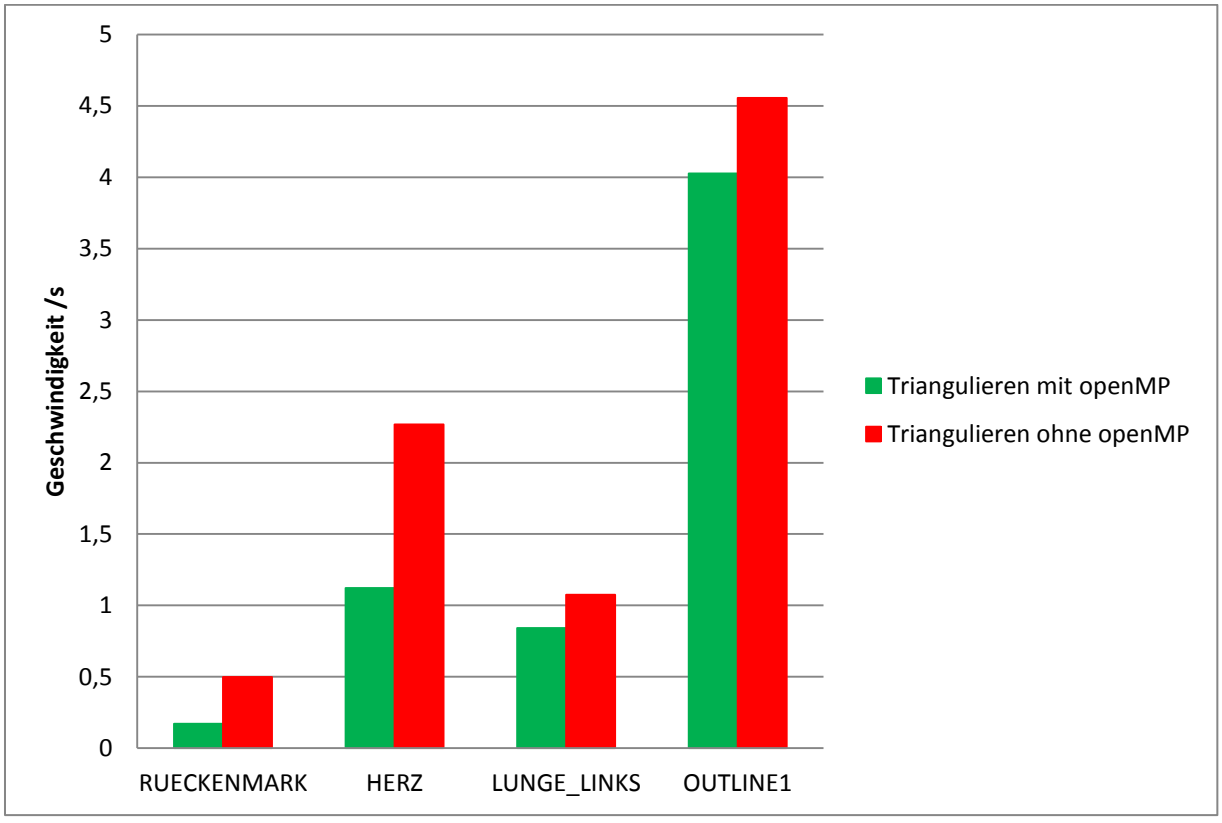

Abbildung 52: Vergleich Triangulierungsdauer zwischen dem im Rahmen dieser Arbeit entwickelten Algorithmus mit und ohne OpenMP Parallelisierung

Das Diagramm in Abbildung 52 zeigt die Laufzeiten der Triangulierung mit und ohne Parallelisierung durch OpenMP. Es zeigt deutlich, dass eine Performanzsteigerung durch OpenMP erreicht wurde. In der zugehörigen Tabelle in Kapitel 8.2 befinden sich außerdem Laufzeiten des Dezimierungsalgorithmus mit und ohne Parallelisierung mittels OpenMP. Diese konnte nur geringfügig beschleunigt werden. Jedoch nimmt den größten Teil der Dezimierung die Oberflächenglättung nach Laplace ein. Dieser wurde im Rahmen dieser Arbeit nicht parallelisiert.

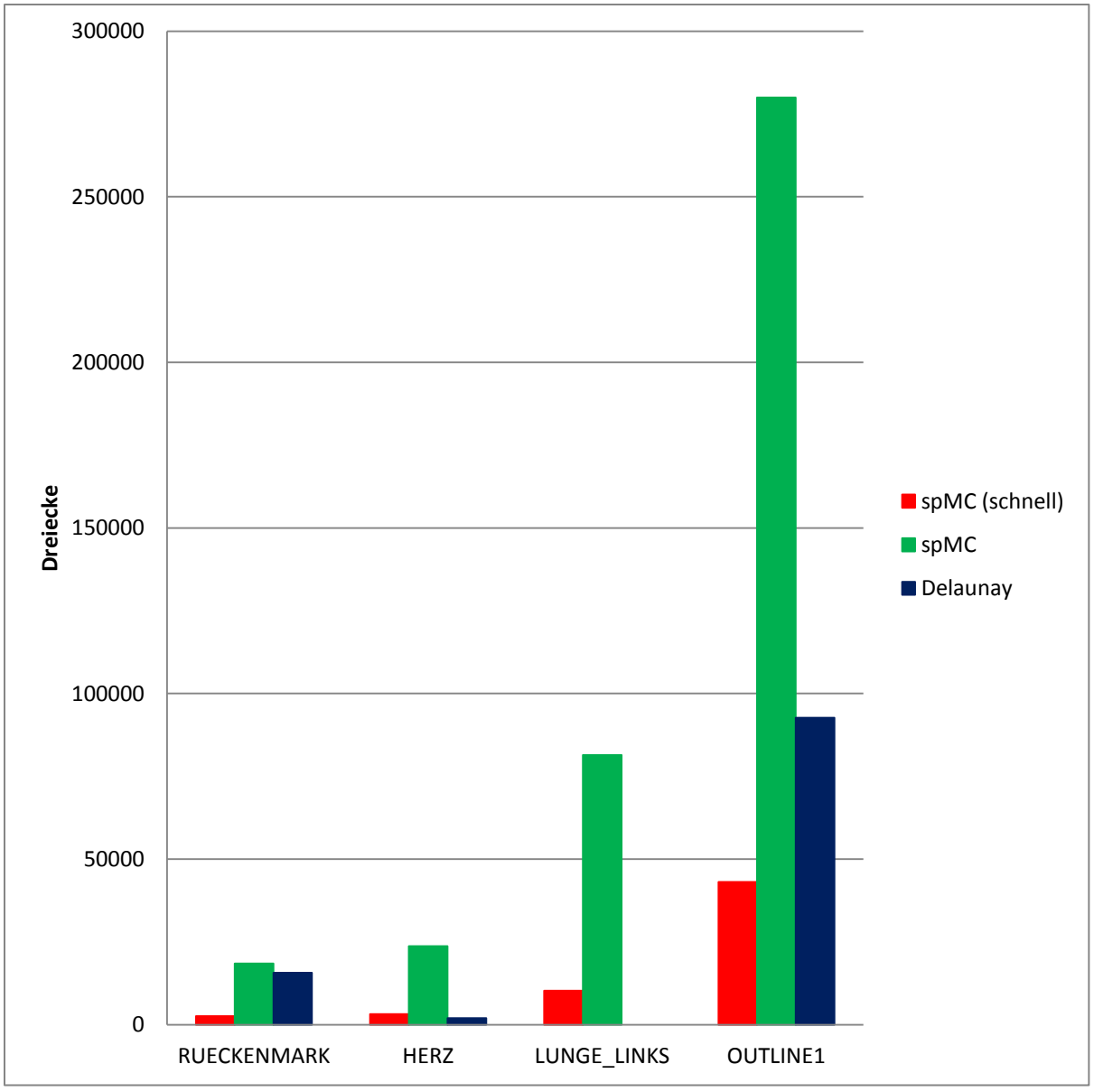

Abbildung 53: Vergleich der Anzahl der von den Algorithmen "spMC (schnell)", spMC und Delaunay erzeugten Dreiecke

Das in Abbildung 53 zu sehende Diagramm listet die von den hier verglichenen Algorithmen erzeugten Dreiecke auf. Wenn man die Anzahl der erzeugten Dreiecke mit dem Volumen der triangulierten Objekte vergleicht, fällt auf, dass die Anzahl der erzeugten Dreiecke beim „spMC" zu der Anzahl der zum Ausfüllen des Volumens benötigten Voxel korreliert. Allgemein erzeugt der „spMC" mit der hier verwendeten Parametrisierung für den Dezimierungsalgorithmus mehr Dreiecke als der „Delaunay". Das Dreiecksoberflächenmodell des „spMC (schnell)" hat deutlich weniger Dreiecke als die zwei anderen Algorithmen.

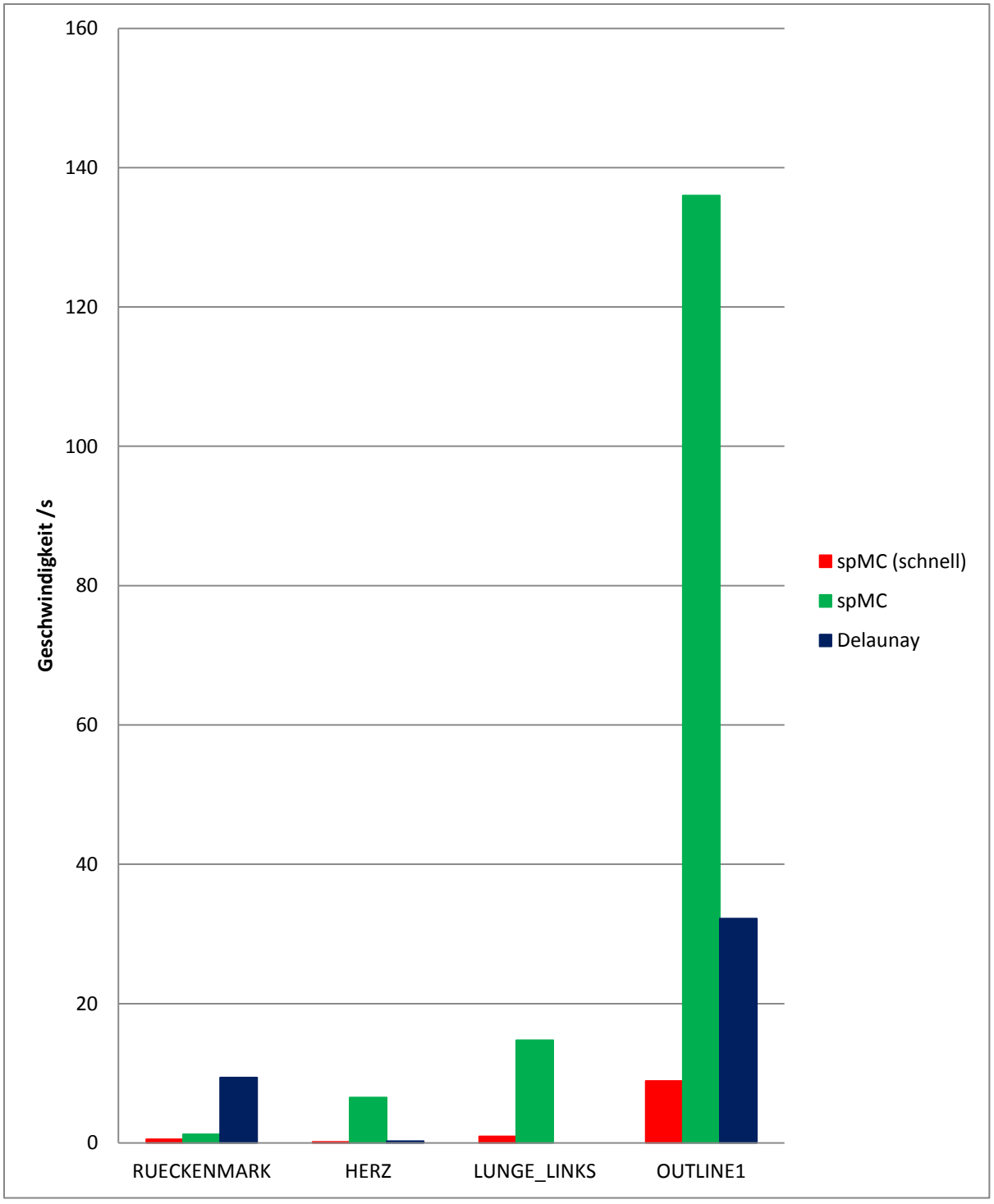

Abbildung 54: Vergleich der Berechnungsdauer von den Algorithmen "spMC (schnell)", spMC und Delaunay

Das Diagramm aus Abbildung 54 vergleicht die Gesammtberechnungsdauer der hier verglichenen Algorithmen. Der „spMC“ schneidet beim „RUECKENMARK“ deutlich besser ab, da dieses viele Konturen beinhaltet, wodurch der „Delaunay“ deutlich mehr Zeit benötigt. Mit steigendem Volumen des zu triangulierten Objekts steigt die Berechnungszeit des „spMC“ deutlich. Die Berechnungszeit des Delaunay für „LUNGE_LINKS“ ist nicht vorhanden, da die Berechnung abgebrochen wurde. Stattdessen wurde die Zeit für die Initialisierung der Datenstrukturen eingetragen.

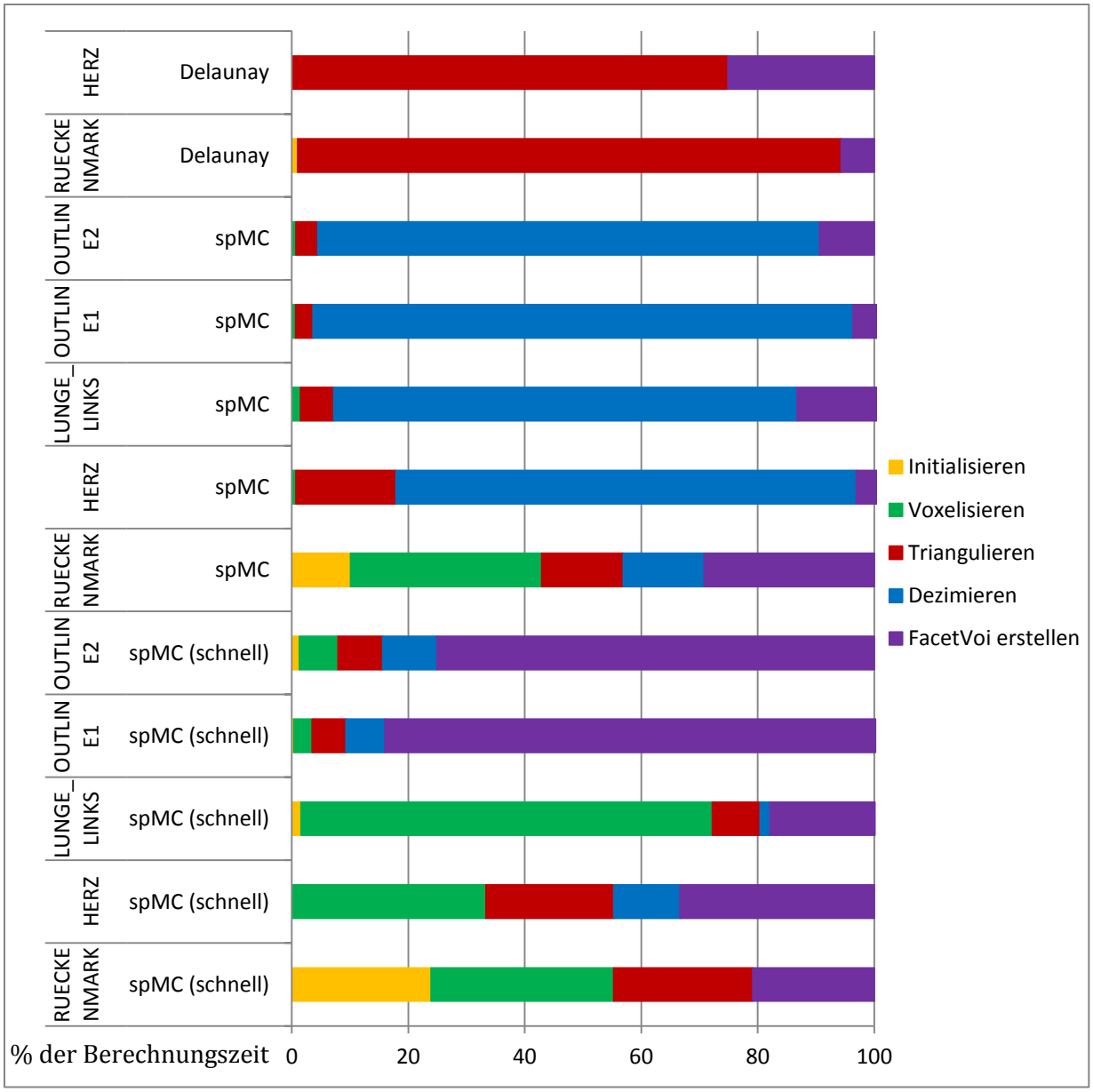

Abbildung 55: Anteil der Einzelschritte an der Gesamtberechnungszeit der Algorithmen "spMC (schnell)", spMC und Delaunay

Das in Abbildung 55 zu sehende Diagramm zeigt die Anteile der Einzelschritte an der Gesamtberechnungszeit der hier verglichenen Algorithmen. Für den „Delaunay" wurden stellvertretend nur zwei Versuche aufgenommen, da die Einzelschritte des „Delaunay" im Rahmen dieser Betrachtung nicht weiter aufgeteilt wurden und somit ein Vergleich nicht sinnvoll wäre. Aus dem Diagramm ist zu erkennen, dass der „spMC" die meiste Zeit für den Dezimierungsschritt benötigt. Mit steigender Oberfläche nimmt diese Berechnungszeit deutlich zu. Der „spMC (schnell)" benötigt viel Zeit zum Füllen der FacetVoi Datenstruktur.

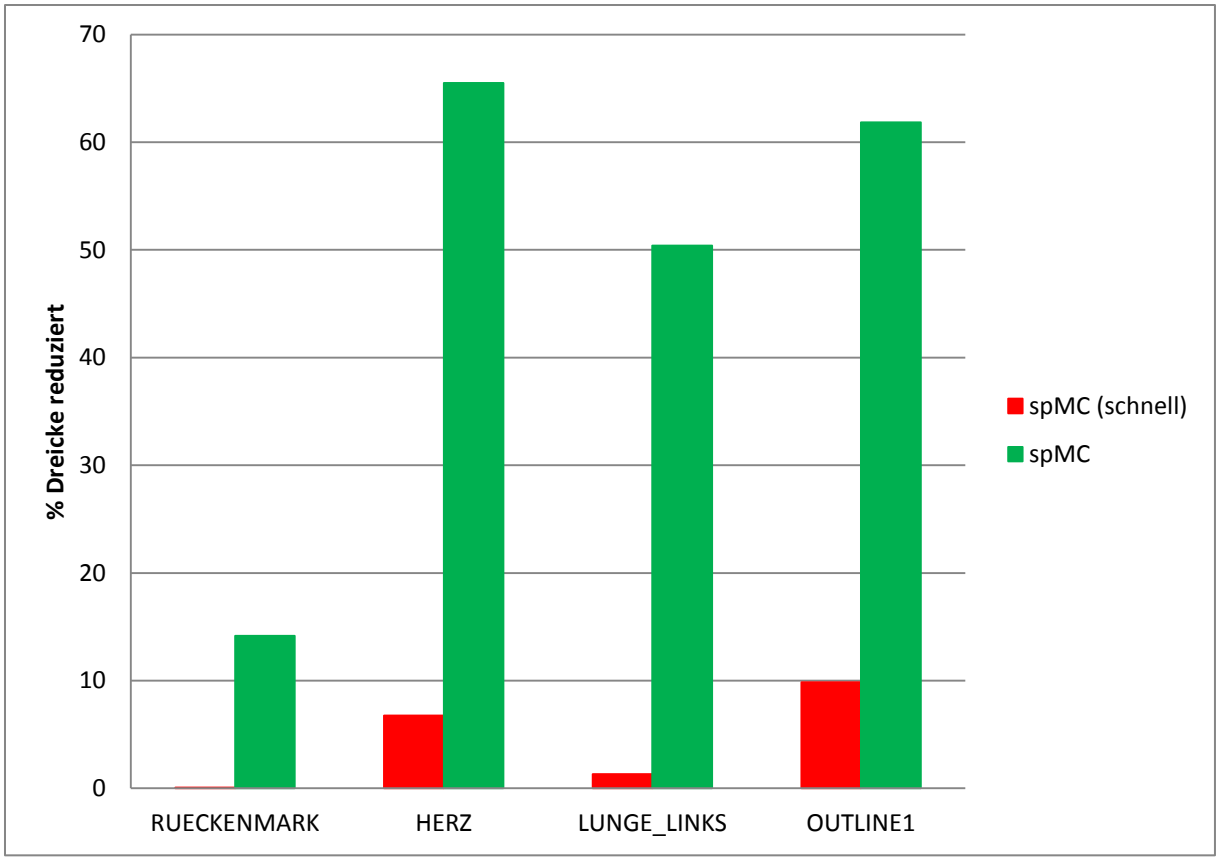

Abbildung 56: Prozent der durch den Dezimierungsalgorithmus dezimierten Dreiecke

Das Diagramm aus Abbildung 56 zeigt wie viel Prozent der von der Triangulierung erzeugten Dreiecke durch die Dezimierung entfernt wurde. Oft kann beim „spMC“ über 50% der ursprünglich erzeugten Dreiecke dezimiert werden. Der „spMC (schnell)“ kann kaum Dreiecke dezimieren, da er so wenig Punkte erzeugt, die nicht auf Konturen liegen und somit dezimierbar sind.

4. DISKUSSION UND AUSBLICK

Das Ziel dieser Arbeit war die Konzeption und Implementierung eines Marching Cube basierten Algorithmus zur punkterhaltenen Oberflächenrekonstruktion aus Konturen. Die Triangulation ist in VIRTUOS ein wichtiges Werkzeug zur Darstellung von VOIs. Die bisher verwendete Implementierung der Delaunay Triangulation hat diverse Schwächen. Deswegen sollte im Rahmen einer Neuimplementierung versucht werden, ein auf dem Marching Cube basierten Algorithmus zu entwerfen.

In diesem Abschnitt werden die Ergebnisse der Entwicklung des neuen Algorithmus diskutiert. Außerdem werden die nicht im Rahmen dieser Arbeit umgesetzten, aber durchaus in Betracht gezogenen, Ansätze zusammengefasst.

4.1 ERGEBNISSE

Im Rahmen dieser Arbeit wurde eine umfangreiche Literaturrecherche durchgeführt. Aus den unterschiedlichen Marching Cube Algorithmen wurde für diese Implementierung der Simplified Pattern Marching Cube [Shi08] ausgewählt. Dieser erzeugt im Verhältnis zu anderen Algorithmen deutlich weniger Dreiecke und bietet eine implizite Lösung der vom Standard Marching Cube bekannten mehrdeutigen Konfigurationen. Durch die in Kapitel 3.1 beschriebene Erweiterung konnte eine punkterhaltene Triangulation erreicht werden. Da keine Lookuptable für den Simplified Pattern Marching Cube vorlag, wurde diese im Rahmen dieser Arbeit erstellt.

Obwohl der Simplified Pattern Marching Cube weniger Dreiecke als der Standard Marching Cube erzeugt, sind es je nach Oberflächengröße immer noch zu viele für eine performante Weiterverarbeitung. Um dieses Problem anzugehen, wurde ein auf dem Dezimierungsverfahren von Schroeder et al. [WJS92] und dem Half-Edge Collapse [Hop93] basierender Algorithmus entworfen, um das Dreiecksnetz auszudünnen ohne die Topologie zu ändern. Gleichzeitig werden die Konturpunkte erhalten. Damit überhaupt ein Dreiecksnetz für die Dezimierung vorliegt, wurde eine für diesen Anwendungsfall optimierte Dreiecksnetzdatenstruktur entworfen.

Da der Simplified Pattern Marching Cube nur benachbarte Voxel verbinden kann, wurde um einen glatten Eindruck in Z-Richtung zu erhalten, das Dreiecksnetz nach Laplace [Her76] geglättet. Dadurch war nach der Glättung die Bedingung verletzt, dass Punkte nur auf einer Schicht liegen dürfen. Deswegen wurde nach der Glättung eine weitere Dezimierung durchgeführt. Alle nicht dezimierten aber geglätteten Punkte wurden wieder zurück auf die ursprüngliche Schicht gezogen.

Der im Rahmen dieser Arbeit entwickelte Algorithmus sollte in seiner Performanz optimiert werden. Dafür wurde die Triangulation und die Dezimierung mittels OpenMP parallelisiert. Es wurde zur Evaluation eine Gegenüberstellung verschiedener Parametrisierungen und der bisherigen Delaunay Triangulation durchgeführt. Diese zeigt, dass durch die Parallelisierung eine deutliche Performanzsteigung der Triangulation erreicht werden konnte. Die Gegenüberstellung der Zeit-Anteile, die ein Schritt im Algorithmus einnimmt, zeigt, dass bei großen Dreiecksnetzen die Dezimierung einen deutlichen Anteil benötigt. Da aus zeitlichen Gründen eine Parallelisierung des Glättungalgorithmus nicht durchgeführt werden konnte, stellt dieser den „Flaschenhals“ des Algorithmus dar. Um ein schnelles Triangulationsergebnis zu erhalten, wurde die Option geboten, die Auflösung des Rasters bei der Voxelisierung zu reduzieren. Dies bewirkt, dass weniger Dreiecke beim Triangulieren erzeugt werden und somit auch weniger dezimiert werden müssen. Diese Option ist besonders für die interaktive Visualisierung während der Segmentierung nützlich, da der Nutzer sofort ein Feedback über seine durchgeführten Schritte erhält.

Aus zeitlichen Gründen wurde ein ausgiebiges Testen mittels Unit Tests ausgeschlossen. Deswegen wurde, für einen visuellen Plausibilitätstest, eine Anbindung an MuPAD/Matlab geschrieben. Diese bietet die Möglichkeit, Strukturen in MuPAD zu definieren und zu exportieren. Das Triangulationsergebnis kann danach wieder in MuPAD importiert und evaluiert werden. Da der Import von großen Dreiecksnetzen jedoch sehr lange braucht, wurde in späteren Entwicklungsschritten eine Anbindung an ParaView umgesetzt. Beide Anbindungen waren für die Entwicklung besonders hilfreich, vor allem als noch keine Integration des Algorithmus in VIRTUOS vorhanden war.

Die Triangulationsergebnisse zeigen, dass der neue Algorithmus eine für die Visualisierung geeignete Oberfläche erzeugt. Im Vergleich zur Delaunay Triangulation erzeugt sie einen detailreicheren Eindruck. Ein Problem des Marching Cube Ansatzes ist, wenn zwei benachbarte Konturen verbunden werden sollen, die stark voneinander abweichen oder komplett disjunkt sind. In diesem Fall erzeugt der bisherige Delaunay Algorithmus wie auch der neue Marching Cube Algorithmus eine klinisch nicht sinnvolle Oberfläche. Die Oberfläche der Marching Cube Triangulation hat zusätzlich den Nachteil, dass prinzipbedingt nur benachbarte Voxel verbunden werden können. Die dafür unter anderem angewannte Lösung mittels Glättung reduziert den Effekt nur teilweise.

4.2 Verwendung des Marching Cube Ansatzes

Das Besondere dieser Arbeit stellt das Unterfangen dar, eine punkterhaltende Triangulation mittels des Marching Cube Ansatzes zu erziehlen, obwohl diese ursprünglich nicht punkterhaltend ist. Dieses Ziel konnte durch zusätzlichen Aufwand realisiert werden.

Im Vergleich zur ursprünglichen Delaunay Triangulation muss jedoch beim Marching Cube eine nachträgliche Dreiecksdezimierung durchgeführt werden. Dies führt zu erheblichen Performanzeinbußen, da der Marching Cube selbst relativ schnell ist. Allgemein lässt sich sagen, dass der im Rahmen dieser Arbeit entwickelte Algorithmus eine gute Marching Cube basierte

Speziallösung für VIRTUOS darstellt. Da jedoch für diese Arbeit das Ziel gesetzt wurde ausschließlich eine Marching Cube basierte Lösung zu entwerfen, wurde kein Vergleich zu anderen Triangulationsalgorithmen gezogen, die punkterhaltent sind. Somit kann im Rahmen dieser Arbeit nicht evaluiert werden, ob dieser Ansatz zu den Bestehenden konkurrenzfähig ist.

4.3 AUSBLICK

Die bisherige Implementierung des im Rahmen dieser Arbeit entwickelten Algorithmus ist der erste erfolgreiche Schritt in die Richtung einer punkterhaltenen Triangulation mittels des Marching Cube Ansatzes. Er hat jedoch noch viel weiteres Potential. Im Laufe der Implementierung haben sich weitere erfolgversprechende Ansätze herauskristallisiert, die aus Zeitgründen nicht in diese Arbeit aufgenommen wurden. Dieses Kapitel schneidet diese kurz an.

In der Diskussion der Ergebnisse hat sich herausgestellt, dass sich der „Flaschenhals“ des Algorithmus in der Oberfächenglättung befindet. Wenn diese parallelisiert wäre, könnte eventuell ein weiterer Performanzzuwachs erreicht werden. Für die Voxelisierung werden zwei verschiedene Voxelisierer verwendet. Um diese zu beschleunigen, könnte ein einziger angepasster Voxelisierer gebaut werden. Dieser könnte im gleichen Schritt die Punktkorrektur vornehmen. Neben der Glättung ist die Dezimierung stark abhängig von der Anzahl der vom Marching Cube erzeugten Dreiecke. Der oft für diesen Ansatz verwendete diskrete Marching Cube [Mon94] ist aber nicht punkterhaltend. Jedoch könnte eine Lösung mittels einer Kombination des adaptiven Marching Cube [Shu95] und des Simplified Pattern Marching Cube [Shi08] möglich sein. Durch den adaptiven Marching Cube könnte zusätzlich die benötigte Glättung minimiert werden, da dieser versucht, Voxel geschickt zusammenzufassen. Dadurch wäre die Auflösung auch nicht mehr begrenzt von der verwendeten Rasterung beim Voxelisieren, da der Raum bei nah beinander liegenden Punkten weiter geteilt wird. Die Datenstruktur der FacetVoi erlaubt es nur zwei Konturschichten mit Dreiecken zu verbinden. Größere Dreiecke sind nicht möglich. Zusätzlich ist es nicht möglich Punkte zu setzen, die nicht auf einer Schicht liegen. Dies erhöht den Aufwand eines Triangulationsalgorithmus deutlich und wäre durch eine geeignetere Datenstruktur leichter zu handhaben. Außerdem könnten durch größere und gleichmäßigere Dreiecke eine bessere Darstellung erzielt werden, da gleichmäßige Dreiecke eine bessere Farbinterpolation beim Shading ermöglichen.

5.QUELLENVERZEICHNIS

[3DS89] 3D Systems Inc. (1989). Stereolithography Interface Specification.

[Ben91] Bendl, R. (1991). *Entwicklung und Implementation einer Benutzerschnittstelle zur virtuellen Strahlentherapiesimulation.* Universität Heidelberg/Fachhochschule Heilbronn.

[Ben93] Bendl, R. (1993). *Methoden zur computerunterstüzten Entwicklung und Anwendung komplexer Bestrahlungstechniken in der dreidimensionalen Strahlentherapieplanung.* Universität Heidelberg.

[Ben13] Bendl, R. (2013). Skript: Diagnoseverfahren - Therapieverfahren - Visualisierung - Registrierung. Hochschule Heilbronn, Ruprecht-Karls-Universität Heidelberg: Studiengang Medizinische Informatik.

[Boi93] Boissonnat, J.-D., Devillers, O., & Teillau, M. (1993). A semidynamic construction of higher-order Voronoi diagrams and its randomized analysis. *Algorithmica 9* , S. 329-356.

[Bor06] Bortfeld, T., Thieke, C., Schlegel, W., & Grosu, A.-L. (2006). Optimization of Treatment Plans, Inverse Planning. In *New Technologies in Radiation Oncology* (S. 207 – 220). Berlin, Heidelberg, New York: Springer.

[Bra06] Braude, I., Marker, J., Museth, K., Nissanov, J., & Breen, D. (2006). Contour-Based Surface Reconstruction using Implicit Curve Fitting, and Distance Field Filtering and Interpolation. *Proc. International Workshop on Volume Graphics* .

[Bru99] Brunet, P., & Scopigno, R. (September 1999). Improved Laplacian Smoothing of Noisy Surface Meshes. *Eurographics '99* , S. 131–138.

[Evg95] Chernyaev, V. E. (1995). Marching Cubes 33: Construction of Topologically Correct Isosurfaces. *Technical Report CERN CN* , 95-17.

[Shi08] Cui, S. H., & Liu, J. (2008). Simplified patterns for extracting the isosurfaces of solid objects. *Image and Vision in Computing , 26*, 174-186.

[Del34] Delaunay, B. N. (1934). Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR , 7* (6), 793-800.

[Han94] Fischer, H.-J. (1994). *Ein objektorientiertes Konzept zur Modellierung, Verwaltung und Manipulation geometrischer Daten und medizinisch/therapeutischen Wissen von anatomischen Strukturen in der dreidimentsionalen Therapieplanung.* Universität Heidelberg/Fachhochschule Heilbronn.

[Gar97] Garland, M., & Heckbert, P. (1997). Surface simplification using quadric error metrics. *Siggraph '97* , S. 209-216.

[Han05] Hansen, C. D., & Johnson, C. R. (2005). In *The Visualization Handbook* (S. 39ff). Elsevier Academic Press.

[Har04] Harder, N. (2004). *Schnelle Triangulation zur Oberflächeninter- und Extrapolation für die schnelle, interaktive Segmentierung anatomischer Strukturen in der adaptiven Strahlentherapieplanung.* Universität Heidelberg/Fachhochschule Heilbronn.

[Her76] Herrmann, L. R. (1976). Laplacian-isoparametric grid generation scheme. *Journal of the Engineering Mechanics Division , 102* (5), 749–907.

[Hop93] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., & Stuetzle, W. (1993). *Mesh optimization.* University of Washington.

[ISO11] ISO/IEC JTC 1/SC 07 Software and systems engineering. (2011). ISO/IEC 25010 - System und Software-Engineering - Qualitätskriterien und Bewertung von System und Softwareprodukten (SQuaRE) - Qualitätsmodell und Leitlinien.

[Lor87] Lorensen, W. E., & Cline, H. E. (July 1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics* , S. 163-169.

[Mon94] Montani, C., Scateni, R., & Scopigno, R. (1994). Discretized Marching Cubes. In *Proceedings of Visualization '94* (S. 281-287). IEEE Computer Society Press.

[FRe04] Reck, F., Dachsbacher, C., Grosso, R., Greiner, G., & Stamminger, M. (2004). Realtime isosurface extraction with graphics hardware. *Proc. Eurographics* , 1-4.

[Sho13] Shontz, S. M., & Nistor, D. M. (2013). CPU-GPU Algorithms for Triangular Surface. In *Proceedings of the 21st International Meshing Roundtable* (S. 475-492). Springer Berlin Heidelberg.

[Shu95] Shu, R., Zhou, C., & Kankanhalli, M. S. (1995). Adaptive Marching Cubes. *The Visual Computer* , 202-217.

[Sym08] *Symbolic Math Toolbox™ 5 MuPAD® Tutorial.* (2008). MathWorks™.

[Tim04] Timothy, S. N., J., B. B., Pavan, E., Amit, N., & Abouzar, D. (2004). High performance SIMD marching cubes isosurface extraction. *Computers & Graphics , 28*, 213–233.

[Tri06] Trisiripisal, P., Lee, S.-M., & Abbott, A. L. (2006). Iterative Image Coding using Hybrid Wavelet-based Triangulation. *Advances in Multimedia Modeling* , S. 309-321.

[WJS92] W., J. S., J., A. Z., & W., E. L. (1992). *Decimation of Triangle Meshes.* SIGGRAPH.

[Wil92] Wilhelms, J., & Van Gelder, A. (1992). Octree for faster isosurface generation. *ACM Trans Graph* , 201-227.

6. ABBILDUNGSVERZEICHNIS

7.FORMELVERZEICHNIS

8.ANHANG

8.1 GLOSSAR

MC	Der Marching Cube [Lor87] Algorithmus wird in Kapitel 2.2.4 näher beschrieben.
spMC	Simplified Pattern Marching Cube [Shi08]
EdgeTable	Eine im Rahmen dieser Arbeit verwendete Datenstruktur, die Vektoren in die Ecken eines Einheitswürfels hält. Siehe Kapitel 3.2.
LUT	Eine Lookuptable beinhaltet vorberechnete Werte, damit diese zur Laufzeit eines Programms nicht neu berechnet werden müssen.
OpenMP	Die Open Multi-Processing Schnittstelle ist ein Standard zur Shared-Memory-Programmierung in C++/C und Fortran.
VIRTUOS	VIRTUal RadiOtherapy Simulator [Ben91] [Ben93] ist eine Software des DKFZ für die Strahlentherapie-Planung
VOI	Eine Volume of interest ist ein Begriff aus der Strahlentherapie, welche eine für die aktuelle Planung wichtige Struktur meint.
VoiManSTL	Das „Volume of interest management standart library“ [Han94] Modul ist Teil von VIRTUOS und dient dem Verwalten von geometrischen Repräsentationen von VOIs. Siehe Kapitel 2.2.3.
Voxel	Der Begriff bezeichnet ein dreidimensionales Pixel und beschreibt somit ein Volumen.
STL-Format	Das Surface Tesselation Language [3DS89] Format ist ein Format um Oberflächen mit Dreiecken zu beschreiben.
Schrotrauschen	Diese auch als Salz & Pfeffer bezeichnete Rauschen ist eine Art statistisches Rauschen.
DiscMC	Discrete Marching Cubes [Mon94]
CPU	Central Processing Unit
GPU	Graphics Processing Unit
Dreiecksnetz	In der Literatur oft in englisch als „Mesh“ bezeichnet.
Konfiguration	Der Begriff meint im Rahmen dieser Arbeit eine bestimmte Kombination von acht markiert bzw. nicht markierten benachbarten Voxeln.
Shading	Shading oder Schattierung ist ein Begriff aus der 3D-Computergrafik, die das Schattieren von Oberflächen bezeichnet. Diese wird mittels eines Interpolationsverfahrens und den auf der Oberfläche liegenden Normalenvektoren berechnet.
UML	Die Unified Modeling Language ist ein Standard zur Spezifikation, Konstruktion und Dokumentation von Systemen. Sie wird von der Object Management Group entwickelt.
CUDA	Die Compute Unified Device Architecture ist eine von der Firma Nvidia entwickelte Technik, Programmteile auf der GPU abzuarbeiten.
OpenCL	Die Open Computing Language ist eine Schnittstelle für uneinheitliche Parallelrechner, die von der Firma Apple entwickelt wurde. Somit können CPUs, GPUs und DSPs zusammen rechnen.
SSE	Die Streaming SIMD Extensions ist eine von von der Firma Intel entwickelte Befehlssatzerweiterung für x86 Prozessor Architektur. Sie beinhaltet Befehle zur Parallelisierung.
MuPAD/Matlab	MuPAD ist ein Computeralgebraprogramm und ist Bestandteil von Matlab. Matlab ist ein von The MathWorks entwickeltes Programm für numerische Berechnungen und Simulation.

Paraview	ParaView ist ein auf VTK basierendes Open-Source Programm für die wissenschaftliche Visualisierung. Es wurde von der Firma Kitware entwickelt.
VTK	Das Visualization Toolkit ist eine Open-Source-C++-Klassenbibliothek für die wissenschaftliche Visualsierung, die von der Firma Kitware entwickelt wurde.
Half-Edge Collapse	Der Half-Edge Collapse wird in Kapitel 2.2.6 genauer beschrieben.
Full-Edge Collapse	Der Full-Edge Collapse wird in Kapitel 2.2.6 genauer beschrieben.
Centroid	Englisch für den geometrischen Schwerpunkt
Interne Kontur	In VIRTUOS bezeichnet eine interne Kontur eine Kontur, die dem Benutzer nicht im Konturstapel angezeigt wird. Sie wird jedoch als interne Struktur verwendet um Operationen wie Oberflächenvisualisierung zu ermöglichen.
Stack	Die Stapel- oder auch Kellerspeicher genannte Datenstruktur hält Daten in einem logischen Stapel. Für den Zugriff gilt das Last-In-First-Out-Prinzip.
HFS	Die head first-supine ist eine Bezeichnung für eine bestimmte Patientenposition in einem DICOM Datensatz. Abbildung 4 demonstriert wie sich ein Patient in dieser Position befindet.
FFS	Die feed first-supine ist eine Bezeichnung für eine bestimmte Patientenposition in einem DICOM Datensatz. Sie entspricht der HFS Position, nur dass die Füße statt der Kopf in Richtung Aufnahmemodalität ausgerichtet sind.
Bändermodell	Ein Bändermodell bezeichnet einen durch Bänder dargestellten Konturstapel.
Koordinatensystem	Siehe Kapitel 2.2.2

8.2 WERTE DES PERFORMANZ VERGLEICHS

Die für alle in diesem Kapitel angegebenen Werte wurden auf einem Desktop-Rechner, mit einem Intel Core i5-2400 @3.1GHz/3,1GHz Prozessor, einer NVIDIA Geforce GT430 Grafikkarte und mit 8GB Arbeitsspeicher, ermittelt. In den Tabellen wird der im Rahmen dieser Arbeit entwickelte Algorithmus als „spMC“ bezeichnet. Die Variante in der jeweils 9 Voxel zusammengefasst werden, wird als „spMC (schnell)“ bezeichnet. Der bisherige Triangulationsalgorithmus wird als „Delaunay“ bezeichnet. Als Abbruchkriterium für den Dezimierungsalgorithmus wurde *continueDecimationCount* auf 10000 gesetzt. Das *continueDecimationPercent* Abbruchkriterium wurde auf 50% gesetzt. Das *decimationCrit* Kriterium für das Dezimieren von internen Konturpunkten wurde auf 1,5 Voxel gesetzt. Für die Messung wurden zwei Datensätze mit vorsegmentieren VOIs verwendet. Die Routinen für die Zeitmessung stammten aus dem Standard C/C++ Header „ctime“. Wenn nicht weiter angegeben, wurde der Algorithmus mittels OpenMP parallelisert.

Die untere Tabelle zeigt die gemessenen Zeiten für die Teilschritte der Triangulierung und die Gesamtzeit. In mit „Abbruch“ markierten Zeilen wurde die Berechnung nach 10 Minuten abgebrochen.

Datensatz	**VOI**	**Algorithmus**	**Initialisieren /s**	**Voxelisieren /s**	**Triangulieren /s**	**Dezimieren /s**	**FacetVoi erstellen /s**	**∑ /s**
thoraxCase4	RUECKENMARK	spMC (schnell)	0,13	0,16	0,13	0,00	0,11	0,52
thoraxCase4	HERZ	spMC (schnell)	0,00	0,05	0,03	0,02	0,05	0,14
thoraxCase4	LUNGE_LINKS	spMC (schnell)	0,02	0,67	0,08	0,02	0,17	0,95
thoraxCase4	OUTLINE1	spMC (schnell)	0,03	0,28	0,52	0,58	7,48	8,88
HNcase3	OUTLINE2	spMC (schnell)	0,03	0,16	0,18	0,22	1,76	2,35
thoraxCase4	RUECKENMARK	spMC	0,12	0,40	0,17	0,17	0,36	1,23
thoraxCase4	HERZ	spMC	0,00	0,05	1,12	4,90	0,47	6,54
thoraxCase4	LUNGE_LINKS	spMC	0,00	0,22	0,84	11,61	2,08	14,74
thoraxCase4	OUTLINE1	spMC	0,05	0,86	4,03	120,05	11,00	135,98
HNcase3	OUTLINE2	spMC	0,03	0,50	2,75	63,21	6,89	73,38
thoraxCase4	RUECKENMARK	Delaunay	0,09	-	8,75	-	0,53	9,38
thoraxCase4	HERZ	Delaunay	0,00	-	0,19	-	0,06	0,25
thoraxCase4	LUNGE_LINKS	Delaunay	0,02	-	Abbruch	-	Abbruch	0,00
thoraxCase4	OUTLINE1	Delaunay	0,02	-	30,90	-	1,29	32,21
HNcase3	OUTLINE2	Delaunay	0,03	-	24,63	-	2,43	27,10

Die untere Tabelle zeigt den Performanz Vergleich von den im Rahmen dieser Arbeit entwickelten Algorithmus mit und ohne OpenMP Parallelisierung.

Datensatz	**VOI**	**Algorithmus**	**ohne OpenMP**		**mit OpenMP**		**Performanzgewinn**	
			Triangulierung /s	**Dezimierung /s**	**Triangulierung /s**	**Dezimierung /s**	**Triangulierung /%**	**Dezimierung /%**
thoraxCase4	RUECKENMARK	spMC (schnell)	0,39	0,00	0,13	0,00	67,95	0,00
thoraxCase4	HERZ	spMC (schnell)	0,11	0,02	0,03	0,02	71,56	0,00
thoraxCase4	LUNGE_LINKS	spMC (schnell)	0,23	0,02	0,08	0,02	66,67	6,25
thoraxCase4	OUTLINE1	spMC (schnell)	1,72	0,59	0,52	0,58	70,06	2,70
HNcase3	OUTLINE2	spMC (schnell)	0,00	0,00	0,18	0,22	0,00	0,00
thoraxCase4	RUECKENMARK	spMC	0,50	0,17	0,17	0,17	65,53	0,00
thoraxCase4	HERZ	spMC	2,27	5,16	1,12	4,90	50,51	5,15
thoraxCase4	LUNGE_LINKS	spMC	1,08	11,73	0,84	11,61	21,65	1,07
thoraxCase4	OUTLINE1	spMC	4,56	126,09	4,03	120,05	11,65	4,79
HNcase3	OUTLINE2	spMC	0,00	0,00	2,75	63,21	0,00	0,00

Die untere Tabelle zeigt wie viele Dreiecke die Triangulierungs-Algorithmen erzeugen. Außerdem beinhaltet sie, wie viele Dreiecke durch das Dezimierungsverfahren entfernt wurden.

Datensatz	**VOI**	**Algorithmus**	**Dreiecke vor der Dezimierung**	**Dreiecke im Ergebnis**	**% Dreiecke reduziert**
thoraxCase4	RUECKENMARK	spMC (schnell)	2571	2569	0,08
thoraxCase4	HERZ	spMC (schnell)	3434	3202	6,76
thoraxCase4	LUNGE_LINKS	spMC (schnell)	10445	10307	1,32
thoraxCase4	OUTLINE1	spMC (schnell)	47857	43139	9,86
HNcase3	OUTLINE2	spMC (schnell)	25350	23368	7,82
thoraxCase4	RUECKENMARK	spMC	21536	18488	14,15
thoraxCase4	HERZ	spMC	68662	23690	65,50
thoraxCase4	LUNGE_LINKS	spMC	164255	81455	50,41
thoraxCase4	OUTLINE1	spMC	733901	279966	61,85
HNcase3	OUTLINE2	spMC	461739	154729	66,49
thoraxCase4	RUECKENMARK	Delaunay	-	15676	0,00
thoraxCase4	HERZ	Delaunay	-	1972	0,00
thoraxCase4	OUTLINE1	Delaunay	-	92682	0,00
HNcase3	OUTLINE2	Delaunay	-	6541	0,00

8.3 SIMPLIFIED PATTERN MARCHING CUBE LOOKUPTABLE

Dies ist die in dieser Arbeit verwendete spMC LUT. Sie ist analog zu einem statischen C/C++ Array formatiert. Jeder Eintrag im Array beinhaltet eine weiteres Array für die jeweils einzusetzenden Dreiecke der jeweiligen Konfiguration. Jede Zahl entspricht dem zugehörigen Index des Vertex in der in Kapitel 3.2 definierten EdgeTable. Die Zahl -1 dient als Füllelement. Jeweils drei Zahlen bzw. Punkte definieren ein Dreieck. Die Punkte sind gegen den Uhrzeigersinn angeordnet und bieten somit die Möglichkeit die zugehörigen Normalenvektoren zu errechnen.

{{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 1, 2, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {3, 0, 1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {2, 3, 0, -1, -1, -1, -1, -1, -1, -1},

{1, 2, 3, -1, -1, -1, -1, -1, -1, -1}, {0, 1, 2, 0, 2, 3, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {1, 0, 4, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {2, 0, 4, 2, 4, 1, 1, 0, 2, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {4, 0, 3, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {1, 3, 4, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {4, 0, 2, 3, 4, 2, 2, 0, 3, -1},

{1, 2, 3, -1, -1, -1, -1, -1, -1, -1}, {3, 4, 1, 2, 3, 1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {5, 1, 0, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{2, 1, 5, -1, -1, -1, -1, -1, -1, -1}, {2, 0, 5, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {5, 1, 3, 0, 5, 3, 3, 1, 0, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {2, 3, 0, -1, -1, -1, -1, -1, -1, -1},

{3, 1, 5, 3, 5, 2, 2, 1, 3, -1}, {0, 5, 2, 3, 0, 2, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 4, 5, -1, -1, -1, -1, -1, -1, -1},

{4, 5, 1, -1, -1, -1, -1, -1, -1, -1}, {1, 0, 4, 5, 1, 4, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 4, 5, -1, -1, -1, -1, -1, -1, -1},

{2, 1, 4, 5, 2, 4, 4, 1, 5, -1}, {5, 2, 0, 5, 0, 4, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {3, 4, 5, 3, 5, 0, 0, 4, 3, -1},

{4, 5, 1, -1, -1, -1, -1, -1, -1, -1}, {1, 3, 4, 5, 1, 4, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {5, 0, 4, 0, 2, 1, 4, 0, 3, -1},

{5, 1, 4, 0, 3, 1, 2, 1, 5, -1}, {3, 4, 5, 2, 3, 5, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{6, 2, 1, -1, -1, -1, -1, -1, -1, -1}, {6, 2, 0, 1, 6, 0, 0, 2, 1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {3, 0, 1, -1, -1, -1, -1, -1, -1, -1},

{3, 2, 6, -1, -1, -1, -1, -1, -1, -1}, {0, 2, 6, 0, 6, 3, 3, 2, 0, -1},

{3, 1, 6, -1, -1, -1, -1, -1, -1, -1}, {1, 6, 3, 0, 1, 3, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {1, 0, 4, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{6, 2, 1, -1, -1, -1, -1, -1, -1, -1}, {4, 1, 0, 1, 6, 5, 0, 1, 2, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {4, 0, 3, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {4, 2, 1, -1, -1, -1, -1, -1, -1, -1},

{3, 2, 6, -1, -1, -1, -1, -1, -1, -1}, {6, 3, 2, 3, 4, 7, 2, 3, 0, -1},

{6, 0, 3, -1, -1, -1, -1, -1, -1, -1}, {3, 4, 1, 1, 6, 3, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{1, 5, 6, -1, -1, -1, -1, -1, -1, -1}, {0, 5, 6, 0, 6, 1, 1, 5, 0, -1},

{5, 6, 2, -1, -1, -1, -1, -1, -1, -1}, {5, 6, 2, -1, -1, -1, -1, -1, -1, -1},

{1, 5, 6, 1, 6, 2, -1, -1, -1, -1}, {2, 0, 5, 6, 2, 5, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{1, 5, 6, -1, -1, -1, -1, -1, -1, -1}, {6, 1, 5, 1, 3, 2, 5, 1, 0, -1},

{3, 2, 5, 6, 3, 5, 5, 2, 6, -1}, {6, 2, 5, 1, 0, 2, 3, 2, 6, -1},

{6, 3, 1, 6, 1, 5, -1, -1, -1, -1}, {0, 5, 6, 3, 0, 6, -1, -1, -1, -1},

{6, 5, 4, -1, -1, -1, -1, -1, -1, -1}, {0, 4, 6, 5, 0, 6, 6, 4, 5, -1},

{1, 4, 6, -1, -1, -1, -1, -1, -1, -1}, {4, 6, 1, 0, 4, 1, -1, -1, -1, -1},

{4, 6, 2, 4, 2, 5, 5, 6, 4, -1}, {4, 5, 0, 1, 2, 5, 6, 5, 4, -1},

{1, 4, 6, 1, 6, 2, -1, -1, -1, -1}, {2, 0, 4, 6, 2, 4, -1, -1, -1, -1},

{6, 5, 4, -1, -1, -1, -1, -1, -1, -1}, {3, 4, 0, 4, 6, 7, 0, 4, 5, -1},

{6, 7, 1, -1, -1, -1, -1, -1, -1, -1}, {1, 3, 4, 4, 6, 1, -1, -1, -1, -1},

{2, 6, 3, 7, 4, 6, 5, 6, 2, -1}, {6, 3, 4, 0, 2, 5, -1, -1, -1, -1},

{1, 4, 6, 6, 3, 1, -1, -1, -1, -1}, {6, 3, 4, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 1, 2, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 3, 7, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {1, 3, 7, 1, 7, 0, 0, 3, 1, -1},

{7, 3, 2, -1, -1, -1, -1, -1, -1, -1}, {0, 2, 7, -1, -1, -1, -1, -1, -1, -1},

{7, 3, 1, 2, 7, 1, 1, 3, 2, -1}, {2, 7, 0, 1, 2, 0, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {7, 4, 0, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {1, 0, 7, 4, 1, 7, 7, 0, 4, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {7, 4, 0, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {4, 0, 7, 3, 2, 0, 1, 0, 4, -1},

{3, 7, 4, -1, -1, -1, -1, -1, -1, -1}, {0, 3, 7, 4, 0, 7, -1, -1, -1, -1},

{3, 7, 4, -1, -1, -1, -1, -1, -1, -1}, {4, 1, 3, 4, 3, 7, -1, -1, -1, -1},

{2, 7, 4, 2, 4, 3, 3, 7, 2, -1}, {0, 2, 7, 4, 0, 7, -1, -1, -1, -1},

{4, 3, 7, 3, 1, 0, 7, 3, 2, -1}, {2, 7, 4, 1, 2, 4, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {5, 1, 0, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{2, 1, 5, -1, -1, -1, -1, -1, -1, -1}, {5, 3, 2, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {0, 3, 7, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {7, 0, 3, 0, 5, 4, 3, 0, 1, -1},

{7, 3, 2, -1, -1, -1, -1, -1, -1, -1}, {7, 1, 0, -1, -1, -1, -1, -1, -1, -1},

{5, 2, 1, 2, 7, 6, 1, 2, 3, -1}, {0, 5, 2, 2, 7, 0, -1, -1, -1, -1},

{5, 4, 7, -1, -1, -1, -1, -1, -1, -1}, {0, 7, 5, -1, -1, -1, -1, -1, -1, -1},

{7, 5, 1, 7, 1, 4, 4, 5, 7, -1}, {0, 7, 5, 0, 5, 1, -1, -1, -1, -1},

{5, 4, 7, -1, -1, -1, -1, -1, -1, -1}, {5, 6, 0, -1, -1, -1, -1, -1, -1, -1},

{1, 5, 2, 6, 7, 5, 4, 5, 1, -1}, {0, 7, 5, 5, 2, 0, -1, -1, -1, -1},

{3, 7, 5, 4, 3, 5, 5, 7, 4, -1}, {7, 5, 0, 3, 7, 0, -1, -1, -1, -1},

{7, 4, 3, 0, 1, 4, 5, 4, 7, -1}, {3, 7, 5, 3, 5, 1, -1, -1, -1, -1},

{2, 7, 3, 7, 5, 6, 3, 7, 4, -1}, {0, 2, 7, 7, 5, 0, -1, -1, -1, -1},

{5, 2, 7, 3, 1, 4, -1, -1, -1, -1}, {5, 2, 7, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{2, 6, 7, -1, -1, -1, -1, -1, -1, -1}, {2, 6, 7, -1, -1, -1, -1, -1, -1, -1},

{1, 6, 7, 1, 7, 2, 2, 6, 1, -1}, {7, 2, 6, 2, 0, 3, 6, 2, 1, -1},

{6, 7, 3, -1, -1, -1, -1, -1, -1, -1}, {0, 3, 6, 7, 0, 6, 6, 3, 7, -1},

{6, 7, 3, -1, -1, -1, -1, -1, -1, -1}, {7, 3, 6, 2, 1, 3, 0, 3, 7, -1},

{2, 6, 7, 2, 7, 3, -1, -1, -1, -1}, {7, 0, 2, 7, 2, 6, -1, -1, -1, -1},

{3, 1, 6, 7, 3, 6, -1, -1, -1, -1}, {1, 6, 7, 0, 1, 7, -1, -1, -1, -1},

{4, 7, 6, -1, -1, -1, -1, -1, -1, -1}, {6, 4, 0, 6, 0, 7, 7, 4, 6, -1},

{4, 7, 6, -1, -1, -1, -1, -1, -1, -1}, {0, 4, 1, 5, 6, 4, 7, 4, 0, -1},

{2, 6, 4, 7, 2, 4, 4, 6, 7, -1}, {6, 7, 2, 3, 0, 7, 4, 7, 6, -1},

{1, 6, 2, 6, 4, 5, 2, 6, 7, -1}, {4, 1, 6, 2, 0, 7, -1, -1, -1, -1},

{3, 6, 4, -1, -1, -1, -1, -1, -1, -1}, {3, 6, 4, 3, 4, 0, -1, -1, -1, -1},

{4, 5, 3, -1, -1, -1, -1, -1, -1, -1}, {3, 6, 4, 4, 1, 3, -1, -1, -1, -1},

{6, 4, 3, 2, 6, 3, -1, -1, -1, -1}, {2, 6, 4, 2, 4, 0, -1, -1, -1, -1},

{3, 1, 6, 6, 4, 3, -1, -1, -1, -1}, {4, 1, 6, -1, -1, -1, -1, -1, -1, -1},

{7, 6, 5, -1, -1, -1, -1, -1, -1, -1}, {7, 6, 5, -1, -1, -1, -1, -1, -1, -1},

{1, 5, 7, 6, 1, 7, 7, 5, 6, -1}, {0, 5, 1, 5, 7, 4, 1, 5, 6, -1},

{2, 5, 7, -1, -1, -1, -1, -1, -1, -1}, {7, 4, 2, -1, -1, -1, -1, -1, -1, -1},

{5, 7, 2, 1, 5, 2, -1, -1, -1, -1}, {2, 0, 5, 5, 7, 2, -1, -1, -1, -1},

{5, 7, 3, 5, 3, 6, 6, 7, 5, -1}, {3, 7, 0, 4, 5, 7, 6, 7, 3, -1},

{5, 6, 1, 2, 3, 6, 7, 6, 5, -1}, {7, 0, 5, 1, 3, 6, -1, -1, -1, -1},

{2, 5, 7, 2, 7, 3, -1, -1, -1, -1}, {2, 5, 7, 7, 0, 2, -1, -1, -1, -1},

{3, 1, 5, 7, 3, 5, -1, -1, -1, -1}, {7, 0, 5, -1, -1, -1, -1, -1, -1, -1},

{4, 7, 6, 5, 4, 6, -1, -1, -1, -1}, {5, 0, 7, 5, 7, 6, -1, -1, -1, -1},

{6, 1, 4, 6, 4, 7, -1, -1, -1, -1}, {1, 0, 7, 1, 7, 6, -1, -1, -1, -1},

{7, 2, 5, 7, 5, 4, -1, -1, -1, -1}, {7, 2, 5, 5, 0, 7, -1, -1, -1, -1},

{2, 1, 4, 2, 4, 7, -1, -1, -1, -1}, {0, 7, 2, -1, -1, -1, -1, -1, -1, -1},

{4, 3, 6, 4, 6, 5, -1, -1, -1, -1}, {0, 3, 6, 0, 6, 5, -1, -1, -1, -1},

{4, 3, 6, 6, 1, 4, -1, -1, -1, -1}, {3, 6, 1, -1, -1, -1, -1, -1, -1, -1},

{3, 2, 5, 3, 5, 4, -1, -1, -1, -1}, {2, 5, 0, -1, -1, -1, -1, -1, -1, -1},

{1, 4, 3, -1, -1, -1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}}

9.EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig, ohne fremde Hilfe angefertigt worden ist.

Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere, außer der im Literaturverzeichnis angegebenen Literatur verwendet habe. Diese Versicherung bezieht sich sowohl auf Textinhalte sowie alle enthaltenden Abbildungen, Skizzen und Tabellen.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

STADT, DATUM

UNTERSCHRIFT